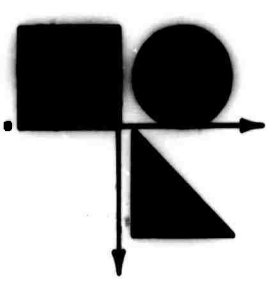
FINAL REPORT - TASK AREA I
(Volume II)

(21 June 1968 - 31 December 1970)

FOR THE PROJECT
RESEARCH IN MACHINE-INDEPENDENT
SOFTWARE PROGRAMMING

Massachusetts
COMPUTER ASSOCIATES
division of
APPLIED DATA RESEARCH, INC.



DISCLAIMER NOTICE

THIS DOCUMENT IS THE BEST QUALITY AVAILABLE.

COPY FURNISHED CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

APPLIED DATA RESEARCH, INC.
LAKESIDE OFFICE PARK, WAKEFIELD, MASSACHUSETTS 01880 · (617) 245-9540

FINAL REPORT - TASK AREA I
(Volume II)

(21 June 1968 - 31 December 1970)

FOR THE PROJECT
RESEARCH IN MACHINE-INDEPENDENT
SOFTWARE PROGRAMMING

Principal Investigators:

Task Area I	Carlos Christensen	(617) 245-9540
Task Area II	Anatol W. Holt	(617) 245-9540

Project Manager:
Robert E. Millstein (617) 245-9540

ARPA Order Number - ARPA 1228

Program Code Number - 8D30

Contractor:	Massachusetts Computer Associates, Inc., Division of ADR
Contract No.:	DAHC04-68-C-0043
Effective Date:	21 June 1968
Expiration Date:	30 September 1971
Amount:	$696,800.00

Sponsored by
Advanced Research Projects Agency
ARPA Order Number - 1228

CA-7102-2612

APPLIED DATA RESEARCH, INC.
LAKESIDE OFFICE PARK, WAKEFIELD, MASSACHUSETTS 01880 · (617) 245-9540

A REPORT ON AMBIT/G
(Volume II)

by

Carlos Christensen
Michael S. Wolfberg
Michael J. Fischer*

CA-7102-2612
February 26, 1971

* Consultant to Applied Data Research, Inc.
Address: Department of Mathematics, M.I.T.,
Cambridge, Massachusetts

This is the second of four volumes of the final report on Task Area I of the project "Research in Machine-Independent Software Programming". This research was supported by the Advanced Research Projects Agency of The Department of Defense and was monitored by U.S. Army Research Office-Durham, Box CM, Duke Station, Durham, North Carolina 27706, under Contract DAHC04-68-C-0043.

CONTENTS

CHAPTER 11
EXAMPLES OF AMBIT/G PROGRAMS

This volume of the report includes eleven complete AMBIT/G programs which demonstrate the various aspects of the language and the Multics implementation of the AMBIT/G System. We begin with three very short programs ('reverse1', 'reverse2', and 'reverse3') to reverse a particular list. These examples include most important aspects of the AMBIT/G language.

The next two examples ('msw3' and 'msw2') are really the same AMBIT/G program, but specified in two different forms: the normal use of rule loading and the exclusive use of data loading. This serves to demonstrate the equivalence of the two methods, and (we expect) motivates the user towards the former method. The program was used during development as a test of the system and it is not an interesting application.

Another example AMBIT/G program ('msw5') is included which was used to test the system. It also is not an interesting application, but it does demonstrate the generality of the function calling mechanism of AMBIT/G.

The remaining five examples are larger programs which perform some interesting application. The program 'lispgc' is a classic example of the use of AMBIT/G which has probably been implemented on every implementation of an AMBIT/G system. It is a LISP garbage collection algorithm presented in Christensen's paper "An Example of the Manipulation of Directed Graphs in the AMBIT/G Programming Language". Another garbage collection program using the same "link-bending" technique is presented as 'mfgarb'.

Next, an interactive program is described ('octdec') which performs conversion of an octal number typed in by the user to the equivalent decimal number which is then typed out. The highlight of this program is the single AMBIT/G rule responsible for performing the arithmetic conversion. This one rule demonstrates the superiority of the AMBIT/G diagram in expressing certain algorithms.

We then present 'quicksort', a program to sort a given list of integers in a recursive manner where the program keeps its own copy of a stack explicitly. This program also includes an example of a "less than" predicate.

Finally, we present a program named 'fact' for computing the factorial of a given integer. The 'factorial' function is composed of two rules where the second rule calls the 'factorial' function recursively. Although the AMBIT/G interpreter does not include the capability for performing recursive function calls, it does have "handles" for augmenting the interpreter to accommodate recursion. The 'fact' program includes a general package of functions for supporting recursion. We recommend that one must have a clear understanding of the function calling mechanism of the interpreter in order to benefit by studying this example, other than the 'factorial' function itself.

This report also includes listings and descriptions of the AMBIT/G interpreter (Vol. III) and AMBIT/G loader (Vol. IV); these programs can also be studied as examples.

OBSERVATIONS

Before presenting the individual examples we wish to include some general observations on the development of the AMBIT/G System and the example programs.

The AMBIT/G System implemented on Multics is a "bug-detecting" system in two senses: first, the interpreter and the built-in functions have been designed to perform considerable checking for errors in the user's program; and second, the system itself continually performs self-checking, and thus system errors are easily detectable. This self-checking attribute is a natural result of the way in which much of the system was written: namely, in AMBIT/G. The checking on the interpreter and loader which is done is essentially the same checking which is done on a user's program. Here we are talking mostly about the concept of a frame in a rule, which serves to mention redundantly much of the data being manipulated.

We conjecture that AMBIT/G is a programming language well suited to writing correct programs since the programmer is continually including redundancy and context in the composition of a rule. Furthermore, the system utilizes that redundancy and context in a debugging environment to perform extensive checking for errors. We have found that errors are often detected within a rule or two of where they were initiated. If AMBIT/G is further developed for a production environment, the redundancy and context will lead to efficient compilation and execution of programs.

It is difficult to measure the effectiveness of AMBIT/G as a language for writing and developing correct programs, particularly in this effort on Multics. Perhaps the implementors were overly careful in their approach to developing this specialized system. Perhaps the Multics environment contributed a lot to this end. Perhaps the extensive design effort on the interpreter before approaching Multics led to a more reliable implementation, but that does not account for the loader and all of the examples. We just can't be sure, but we certainly were impressed at the speed with which bugs were detected and corrected. We should not forget to consider the effectiveness of the two debugging aids which were used in the development: the 'debug' facility of Multics for symbolically debugging PL/I programs, and 'agd', the symbolic debugger of the AMBIT/G System.

There has been, however, a price to pay for the redundancy, namely high computing cost and rather slow execution. But here we did not expend the effort to analyze the running AMBIT/G System. This activity is allegedly one of the supported features of Multics, and we would like to pursue such a study if we continue to develop the system. Once we are highly confident of the reliability of the interpreter and loader, these two AMBIT/G programs can be translated into an optimized form of a PL/I program rather than the current stylized redundant form.

Our aims for the Multics implementation were <u>not</u> for an efficient system, but for a demonstrable and reliable one; these aims were met. If we had sought to produce an efficient system, it might not have been finished.

One drawback of the slow speed of the system is that we decided not to pursue the possible bootstrap of running the AMBIT/G interpreter as a user program. In any case, before attempting a bootstrap run it will be necessary for us to solve certain other problems in the system.

At least, the speed of the system is proportional to its use. Namely, the null complete AMBIT/G run takes less than 10 CPU seconds or costs less than $1.15 during the prime shift of Multics.

A major inefficiency in the operation of the system has been the inability to correct an error detected by the loader and resume execution. We often have to restart from the beginning an AMBIT/G run which has an error in the loader input. Once a program is loaded, however, it can be saved along with the rest of the AMBIT/G Data Graph. Then if an error is detected in the execution of that program, it may be possible to repair the damage and resume or restore the saved data and patch that before resuming.

INTRODUCTORY EXAMPLES: REVERSING A LIST

We present three independent programs which each perform the same function: to reverse a particular list of 'char' nodes connected by nodes of type 'c'. These examples are meant to serve a tutorial purpose of introducing the most important aspects of the AMBIT/G language. These examples will, therefore, be discussed in greater detail than the remaining ones.

All three examples have an initial data configuration of a pointer node 'p x' pointing at a list of four characters: 'P', 'O', 'T', and 'S'. Each example has a different mechanism for ending the list; but they concur in the sense that some particular node is used as an end-of-list indicator, and 'p y' is pointing at that indicator. The given list is forwardly linked, and the purpose of the programs is to end up with 'p y' pointing to a list of the four characters in reverse order: 'S', 'T', 'O', and 'P'. Also 'p x' will be pointing at the end-of-list indicator. The modified list will consist of the same connector nodes with modified connecting links.

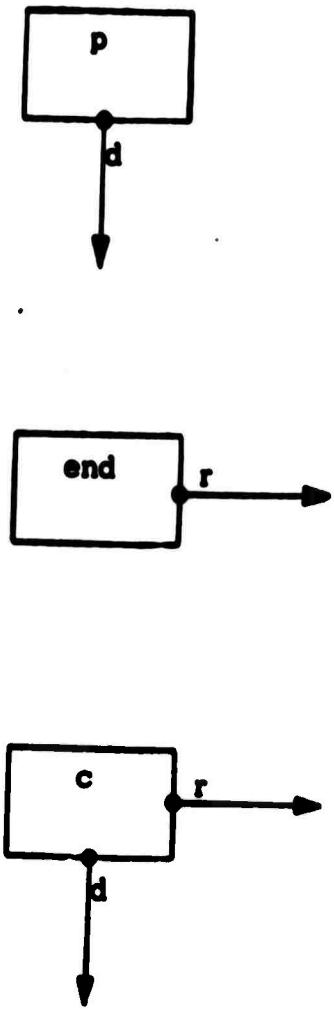
The reversal of the list is accomplished by essentially one AMBIT/G rule which is executed repeatedly until the end of the list is detected. 'p x' is used to "walk" along the given list, one connector node at a time; and the connector node it is pointing to is pushed down on the new list developed at 'p y'.

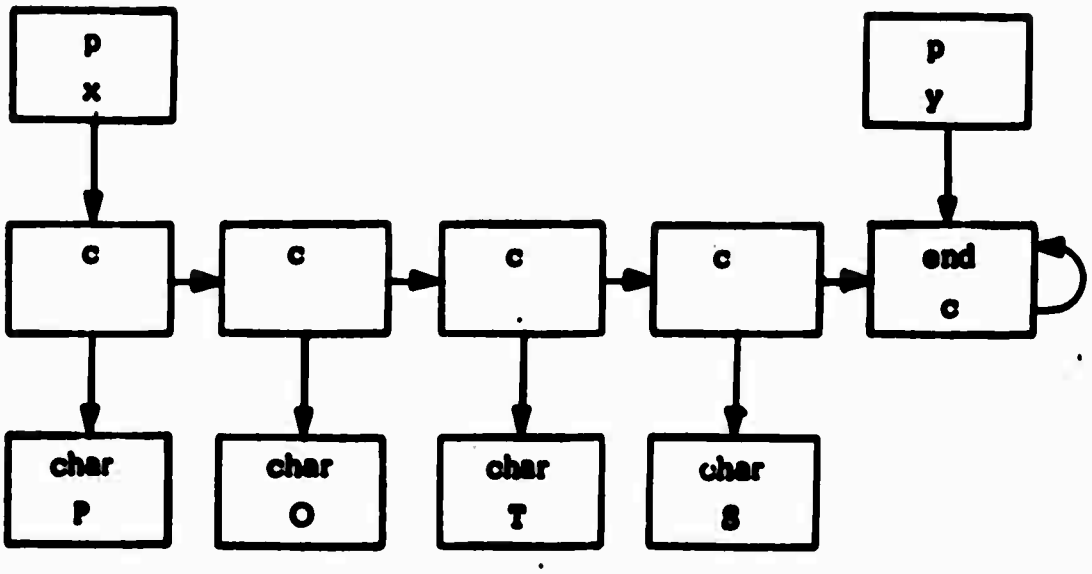
First Version

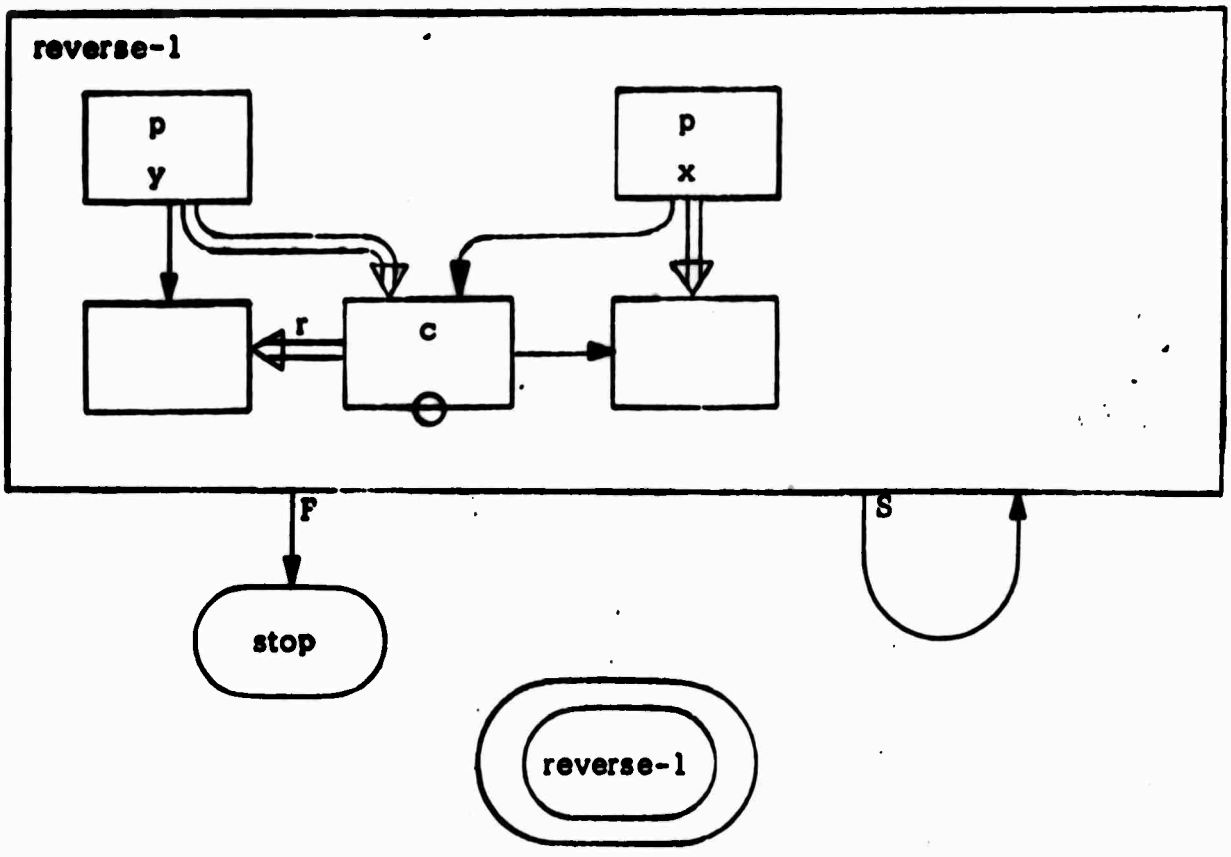
The first example consists of three pages of AMBIT/G diagrams. They are labelled 'r1-1', 'r1-2', and 'r1-3' in the upper right corner of each page. The first page indicates the name of this program is 'reverse1'; the page includes three occurrences of link definitions. First, nodes of type 'p' have a link named 'd'. The large black dot where that link specification is attached to the generic box with a type of 'p' indicates this is a characteristic origin. Thus, there will not be any need to label links in the diagrams which follow when they emanate from a box of type 'p' and attach in the general area of the center of the lower edge of the box. Similarly, two more link definitions are given on page 'r1-1' with characteristic origins. The choice of the names of node types and link names are made by the programmer. Here, 'p' is a mnemonic for "pointer", 'end' is for the "end-of-list indicator",'c' is for "connector", 'd' is for "down", and 'r' is for "right".

On the next page is a diagram of boxes connected by arrows. Note that a large box does not surround the collection of boxes; therefore, we recognize this is a specification of data (rather than a rule). Boxes in data specifications must at least have a type and may have a subname. Note the four nodes of type 'c' without subnames. Arrows which are drawn as part of a data specification must be normal single-line arrows (in contrast to other kinds of arrows allowed in rules). The arrows represent a setting of links in the AMBIT/G data. The arrows do not require labels in this context since they all emanate from characteristic origins. This page does not call for the creation of any data since all nodes exist for all time in a particular AMBIT/G run and are neither created nor destroyed. The mention of a node with no subname represents a unique individual of its type which will never lose its unique identity. As long as it is connected (not necessarily directly) in the data to a

reverse1




p
x
p
y
c
c
c
c
end
c
char
P
char
O
char
T
char
S


reverse-1
p
y
p
x
r
c
F
S
stop
reverse-1

fully named node, it can serve a useful purpose, but if it is ever disconnected completely it will become permanently inaccessible.

Since an AMBIT/G page is accepted by the loader and processed "simultaneously", the first page where links were defined had to be separate from the second page. However, the third page could have been merged with either the first or second pages. It was made a separate page for an arbitrary aesthetic reason.

The third page of this program includes an example of a rule labelled 'reverse-1'; note the large box surrounding a collection of nodes. This rule box has two arrows representing links for program flow emanating from its bottom edge. The 'success' link labelled with 'S' indicates flow of control should return to the very same rule if the interpretation of the rule results in taking the success exit. The 'fail' link labelled 'F' indicates where to go as a fail exit of the rule. The "squashed circle" labelled 'stop' is a rule reference which is the commonly used method of indicating flow of control to a rule which is not represented on the same page. In this case, 'rule stop' is a built-in rule of the system which has the obvious interpretation of terminating the AMBIT/G run.

The contents of the rule 'reverse-1' consists of five boxes connected by six arrows. Three kinds of boxes are included: fully named ('p x' and 'p y'), unnamed (lower left and lower right ones), and a type test box (in the middle). The fact that it is being tested is indicated by the circle along one of its edges. The only kind of box which may have such a circle is a box (in a rule) where only a type is given. The other test mechanism in a rule is a circled arrow, but this rule does not include such an arrow. It includes three single-line arrows specifying the frame of the rule and three double-line arrows specifying links to be modified if the test in the rule succeeds. Note that five arrows emanate from characteristic origins, but the double-line arrow pointing towards the left requires a label.

The doubly encircled 'reverse-1' on page 'r1-3' is a special notation that this is the end of AMBIT/G diagrams to be loaded as one load, and interpretation should begin at rule 'reverse-1'.

Rule interpretation might* proceed as follows whenever interpreter control reaches rule 'reverse-1':

> First frame the data graph as follows: Select 'p y' , follow the 'd' link, and call its destination n1. Select 'p x', follow the 'd' link, and call its destination n2. Select n2, follow the 'r' link, and call its destination n3.
>
> Next test the data graph as follows: Is n2 of type 'c'? If the answer is "no", take the fail exit from the rule.
>
> If the fail exit was not taken, modify the data graph as follows: Select 'p y' and set its 'd' link to point to n2. Select n2 and set its 'r' link to point to n1. Select 'p x' and set its 'd' link to point to n3. Finally, take the success exit from the rule.

Note that this program detects the end of the list by running into a node of type other than 'c'. Since the final interpretation of the rule matches n2 to 'end c', the node 'end c' was designed to have an 'r' link so that the frame of the rule could be applied successfully. This form of list representation is rather contrived and not generally recommended. The other two versions will improve upon this.

This program was also used as the example in the chapter on the AMBIT/G loader (in Vol.I of this report) to demonstrate the encodement of diagrams into text which the loader can accept. Similarly, the hints associated with this program were listed as a sample hint file in the chapter on initialization (also in Vol.I).

As a confirmation that this program performs according to our expectations and as an example of the use of the AMBIT/G debugger, included here is an actual listing of an AMBIT/G run of 'reverse1'. Note the run required about 69 seconds of CPU time (including the loading of the

---

* The word "might" is used because the total ordering given in the three paragraphs of rule interpretation is not required. The commands may be interpreted in any order within a given paragraph provided that each dummy variable (like n1) is associated with a node in the data graph (by a "call" clause) before it is referenced (by a "select" clause).

of the AMBIT/G System and its loading the user program). Incidentally, the use of 'agd' required 1.736 seconds. Small arrows have been drawn in the listing to indicate those lines which were typed by the user. This convention will be used throughout all of the examples.

```
→ hmu

   Multics 13.1a, load 11.0/41.0; 11 users

   r 2031   .502   8↔42

→ ambitg reversel
   AMBIT/G


   r 2034   69.003   85↔648

→ agd

→ p x
   p x:
                d/end c
→ p y
   p y:
                d/c &1105
→ &1105
   c &1105:
                r/c &1103
                d/char S
→ &1103
   c &1103:
                r/c &1101
                d/char T
→ &1101
   c &1101:
                r/c &1099
                d/char O
→ &1099
   c &1099:
                r/end c
                d/char P
→ /q
   r 2035   1.736   55↔39
```

Second Version

In this version, 'reverse2', the end-of-list indicator is a node of type 'c' with subname '**' (borrowed from the AMBIT/L programming language). This convention has forced the one rule of 'reverse1' to be split into two rules (see page r2-3). This has introduced further examples of the kinds of boxes and arrows which may appear in AMBIT/G diagrams. The first rule includes a circled arrow which represents a link to be tested (i.e., does the 'd' link of 'p x' point to 'c **'?) .

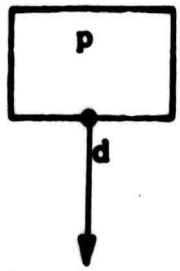
The second rule box on page 'r2-3' does not have a label. This convention is permitted since connectivity of 'success' and 'fail' links among rules is what is important. If such a link is to lead to a rule on a different page, then a rule reference should be used, and that destination rule must have a label.

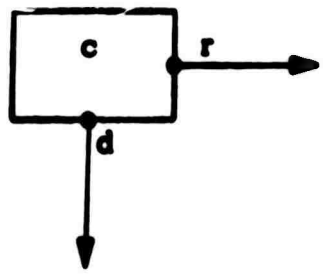
Note the exit arrow from the second rule does not have an 'S' or 'F' label. This indicates there is no 'fail' exit and the given exit is the 'success' exit. In the occasional case when an individual arrow can be used both as a 'success' and 'fail' exit, it can be labelled 'SF'.

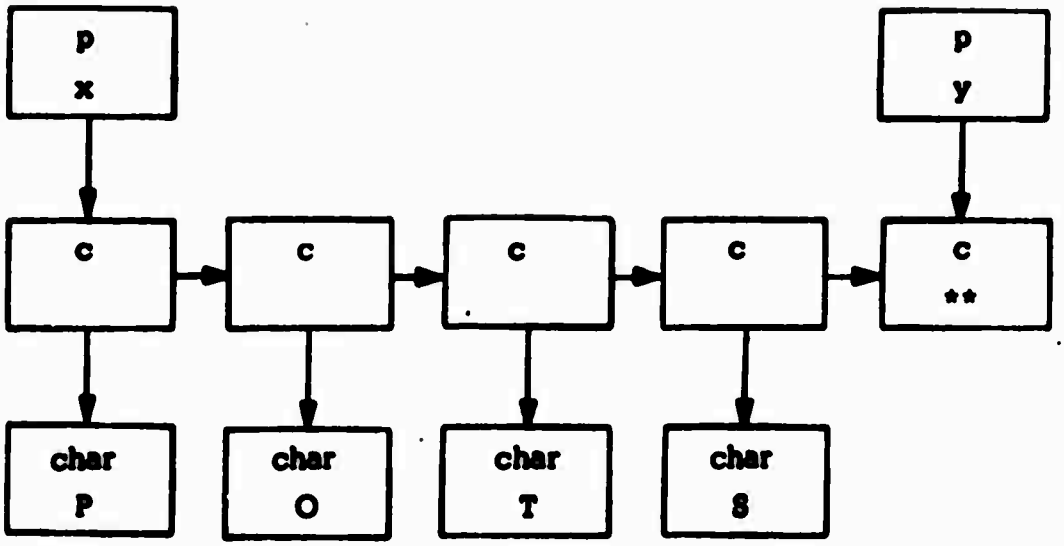
One more kind of object found is this program which was not in the first one is a box named by just a type (with no subname). The interpretation of this object in the second rule on page 'r2-3' affects the matching of the frame. Namely, the rule asserts that 'p y' points down (via link 'd') to a node of type 'c'; that 'p x' points down to a node of type 'c'; and that node in turn points right to a node of type 'c'. If any of these assertions is violated during the attempt to match the frame, an error condition is signalled by the AMBIT/G interpreter and execution is aborted.

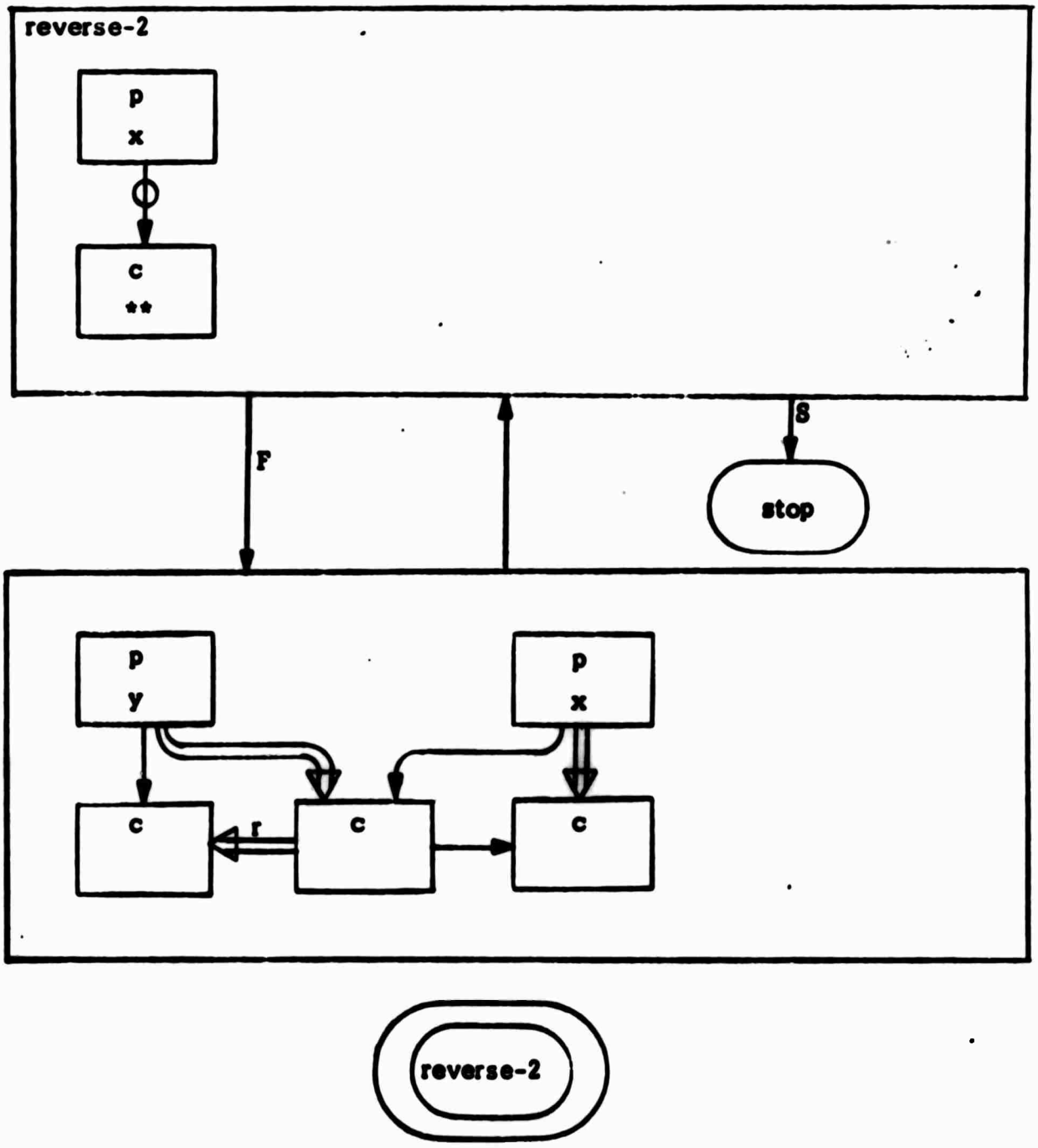
Note that since the test for the end of the list is made in a separate rule, it was not necessary to initialize the 'r' link of 'c **'.

reverse2






p
x
p
y
c
c
c
c
c
**
char
P
char
O
char
T
char
S


reverse-2
p
x
c
**
F
S
stop
p
y
p
x
c
r
c
c
reverse-2

Third Version

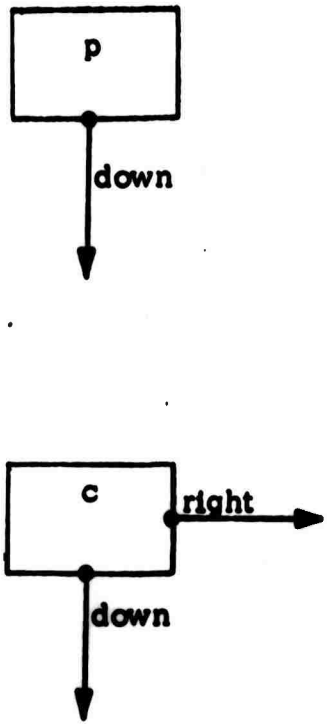
In an attempt to use 'c **' as an end-of-list indicator and to perform list reversal in only one rule, the 'reverse3' program was written. Furthermore, it includes the use of both built-in and user functions. To demonstrate the freedom of choosing any link names, page 'r3-1' includes link definitions for 'down' and 'right' rather than 'd' and 'r'.

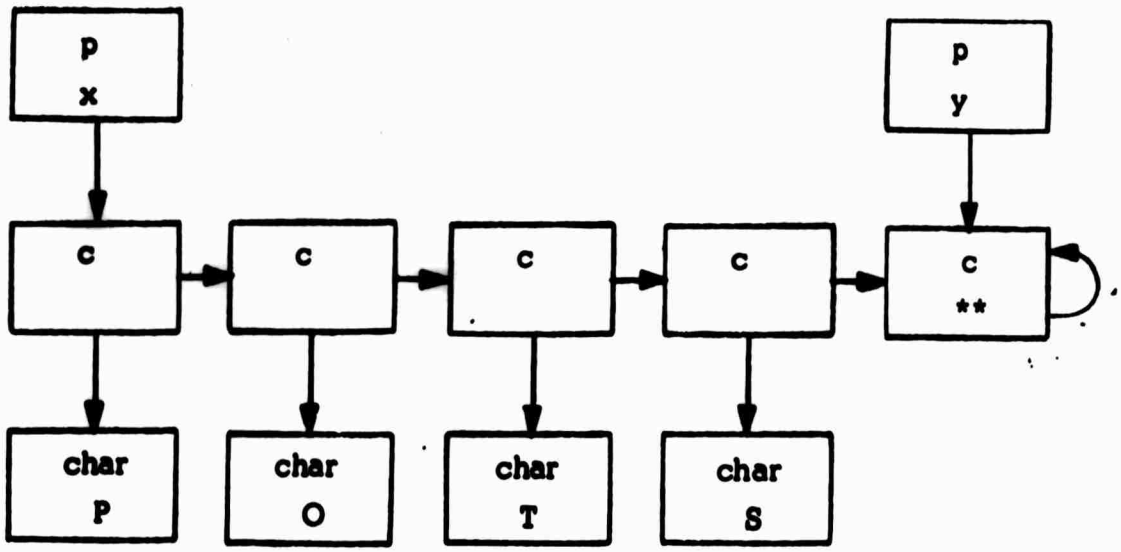
The first rule on page 'r3-3' sets up an argument for the function call in the second rule. That call on the built-in 'read_function' causes the definition of a frame or test use of the link ' ¬= ' with any two tail arguments to be the user function beginning at rule ' ¬= '. Since the two-tailed, one-headed arrow of the second rule is composed of double-lines, it indicates a write call on the built-in 'read_function'.

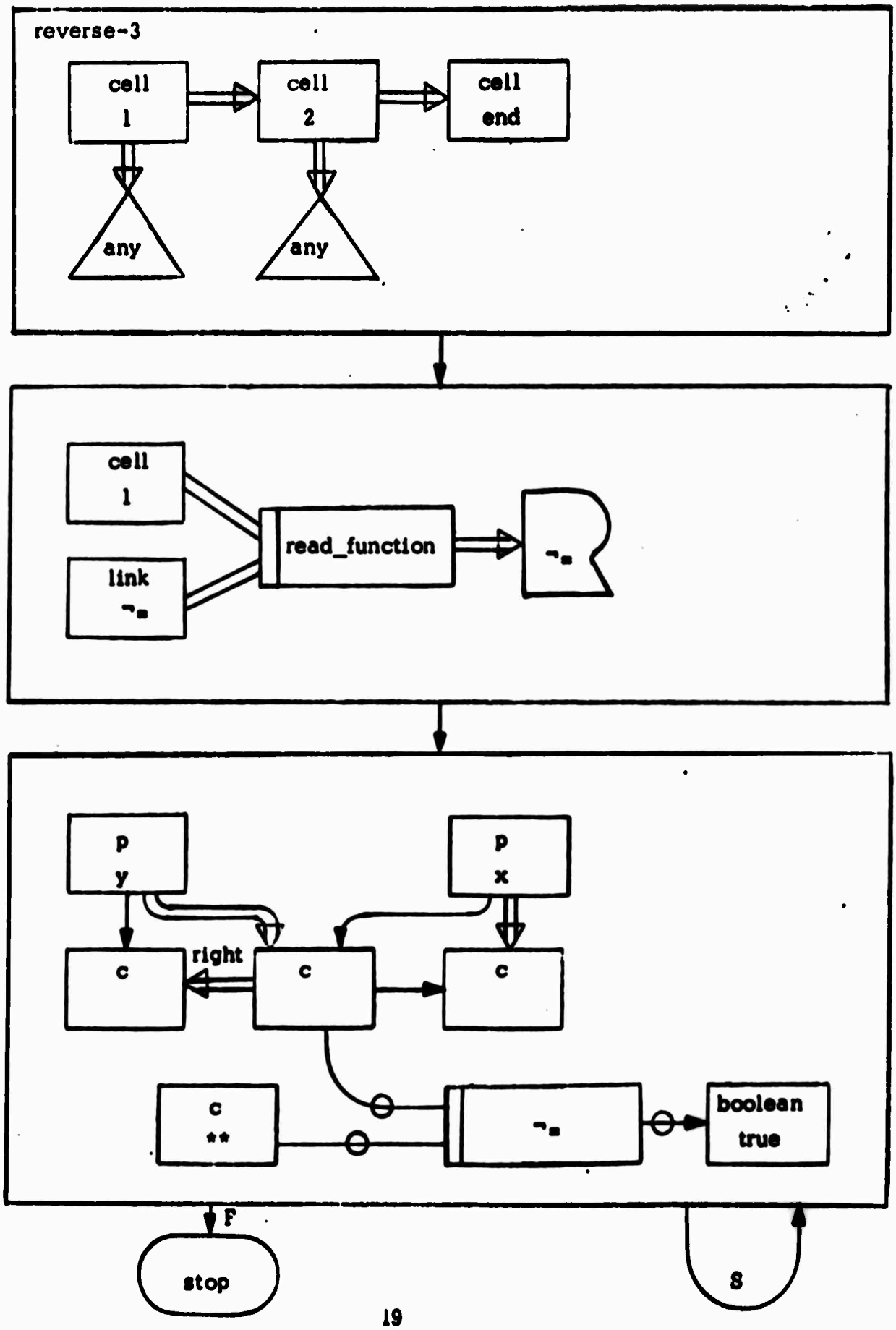
Note the first rule includes boxes with type 'cell'; this is one of the types of nodes built-in to the AMBIT/G System. More significant, however, is the use of a box which is not the standard rectangle. This is a characteristic shape in the same spirit that characteristic origins are used. Namely, the shape of a box can be used to indicate its type. In rule 'reverse-3', the triangular box indicates its type is 'flag', another built-in type. Some of the other built-in types have built-in characteristic shapes; these are given in the beginning of the listing of the interpreter in Vol. III. The second rule on page 'r3-3' includes a box of type 'rule' drawn as an empty capital 'R'. The programmer may define his own characteristic shapes for use in his program. He may optionally use the standard rectangle with a textual type in place of either his own or built-in characteristic shapes.

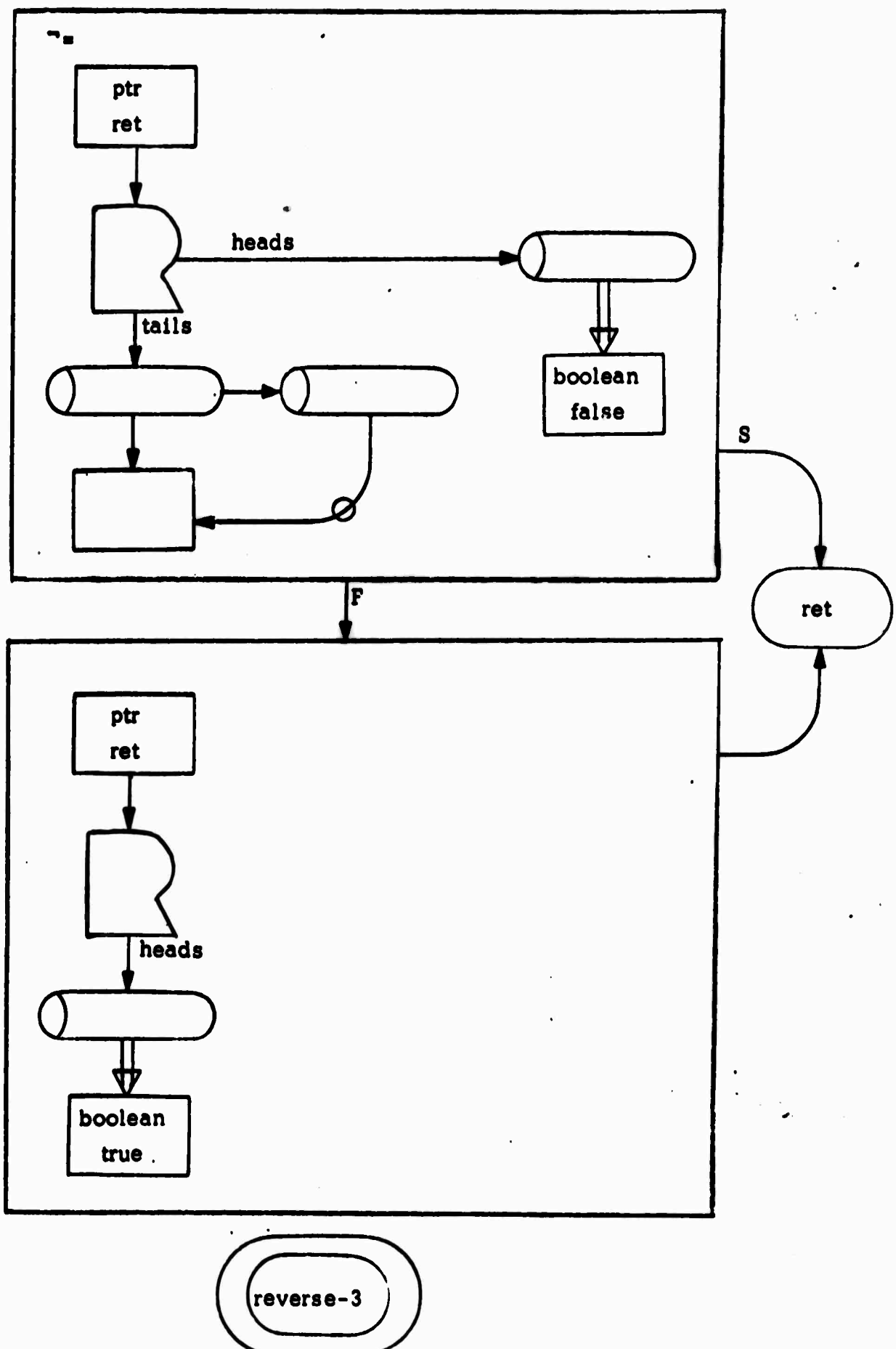
The third rule on page 'r3-3' includes another instance of a box representing a built-in node: 'boolean true'. It also includes a function call on the ' ¬= ' function with a test of the result; this is indicated by the two-tailed, one-headed arrow composed of circled lines.

reverse3




p
x
p
y
c
c
c
c
c
**
char
P
char
O
char
T
char
S


reverse-3
cell 1
cell 2
cell end
any
any
cell 1
link ¬■
read_function
¬■
p y
p x
c
right
c
c
c **
¬■
boolean true
F
stop
S


ptr
ret
heads
tails
boolean
false
S
F
ret
ptr
ret
heads
boolean
true
reverse-3

The ' ¬= ' ("not-equal") function produces a result of 'boolean true' if its two tail arguments are not the same node; the result is 'boolean false' otherwise. Page 'r3-4' includes the two rules (and one rule reference) which constitute the definition of the function. All of the named nodes, labelled links, and characteristic shapes are built-in. The rule named 'ret' is the built-in rule to which control should flow to effect the return from a function call.

The two rules on page 'r3-4' demonstrate the method for obtaining tail arguments and returning a result. Details of the function calling mechanism are described in the section on user-defined functions in the chapter on the interpreter in Vol. I; the fundamental point which should be noted, however, is the role of the function call as a sort of generalized link. That is, in the last rule on page 'r3-3', the function call on '¬=' looks like a test link with two origins (both 'c' nodes), with link name '¬=', and with destination 'boolean true'.

TWO FORMS OF INPUT

The program to be discussed in this section was used during development as a test of the system, especially the loader. As such it demonstrates a few more features of AMBIT/G which have not yet been covered by the examples.

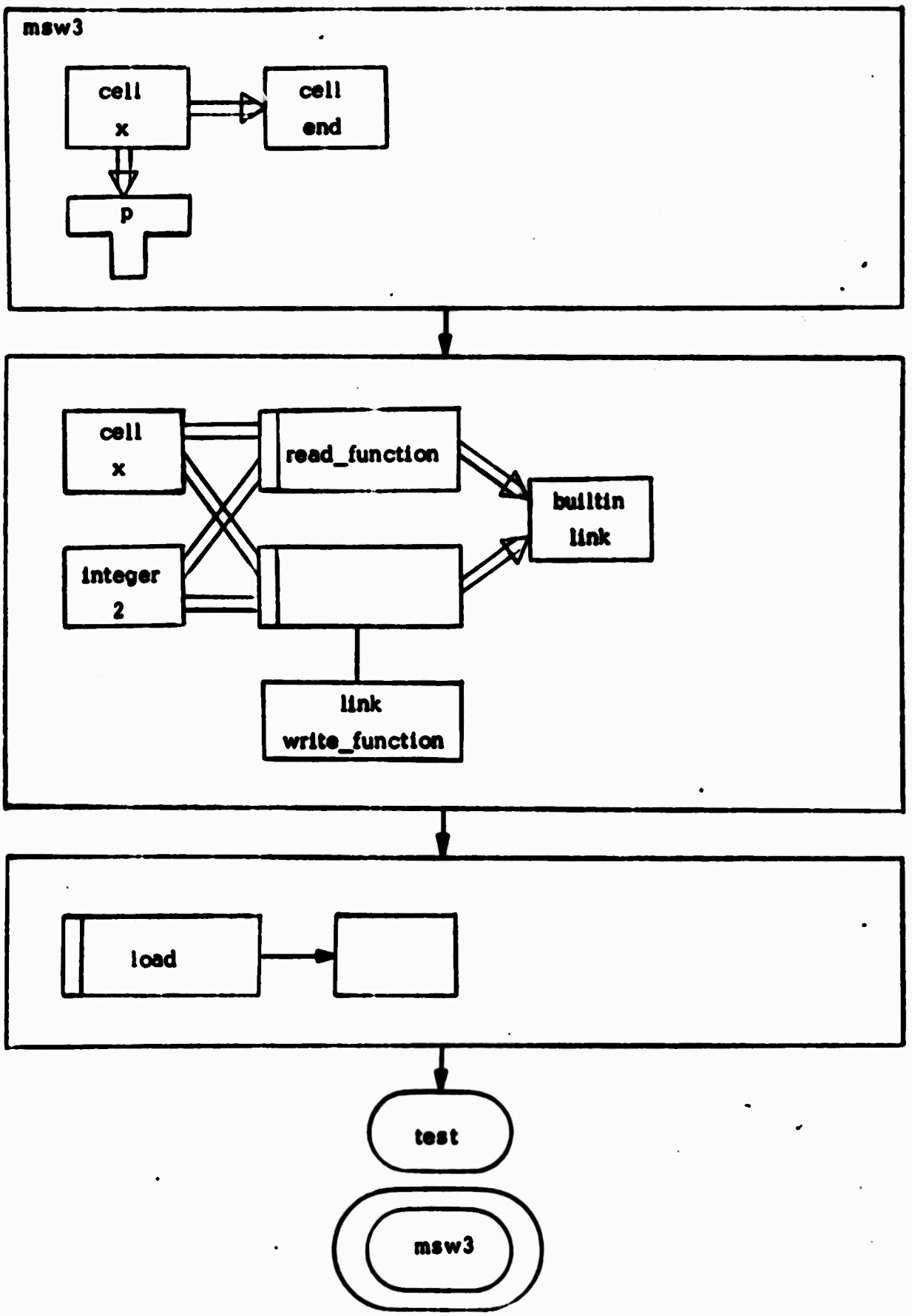
This program 'msw3' is composed of four loader pages. The first page includes three rules and an indicator to start execution at the first of those three rules. The third rule makes a call on the loader to load the remaining three pages. Since the 'success' exit of the third rule on the first page is taken after loading is complete, control can flow to rule 'test' which is on the second page. Also note the returned result (starting rule) is ignored by the user call on the loader; the fourth page has an indicator for starting at rule 'error' but that returned result will be ignored.

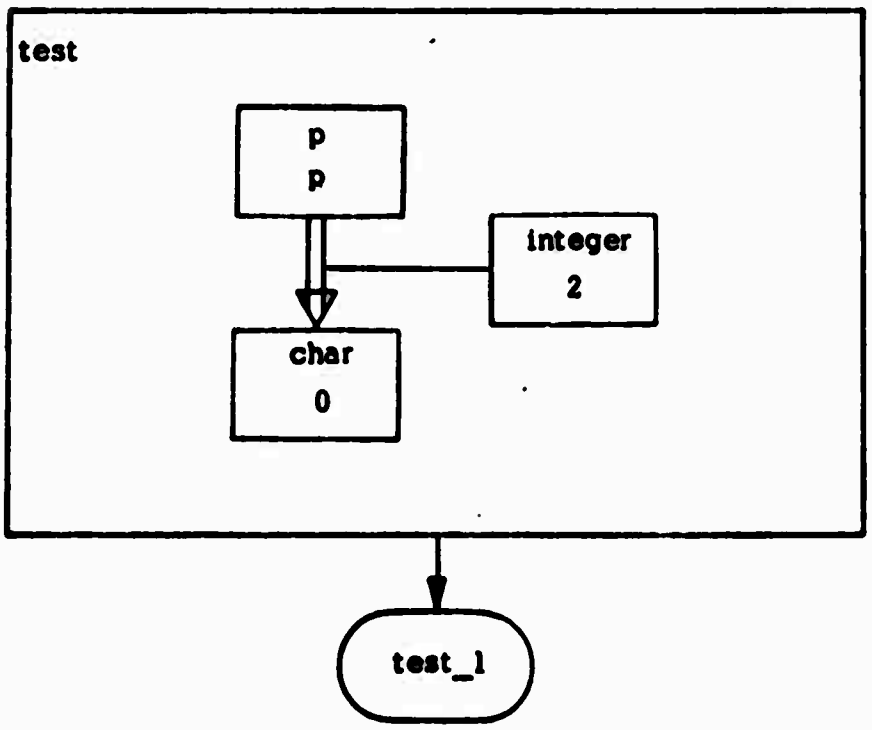
The second rule of 'msw3' demonstrates how a program can define a link for reading and writing. Note, first, the link name is 'integer 2' which is not of type 'link'. Although, it is customary to use 'link' type link names, AMBIT/G has no such restriction. However, it does require the use of an explicit spur when referring to such a link in a rule; see the remaining pages of the program. Secondly, note the use of an explicit spur in the second rule on the write call on the built-in 'write_function'. Since the type of the node at the end of the spur is 'link' this form is optional here, but this form would otherwise be required.

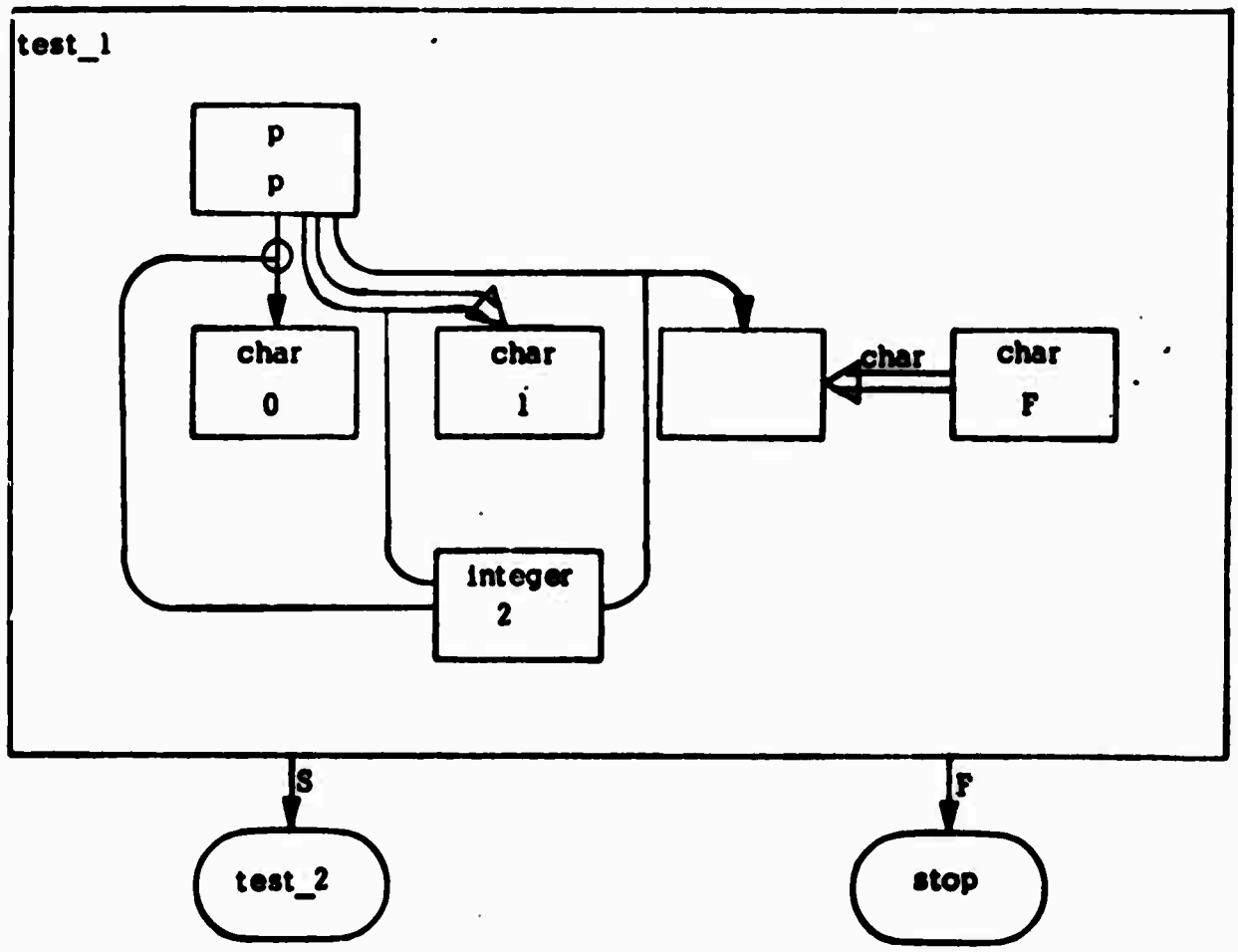
The third page (named 'second rule') shows three kinds of arrows emanating from 'p p'. Also shown is a write call on the built-in 'char' with a tail argument of 'char F', which outputs immediately one character to the terminal. In this program, that character is '0'.

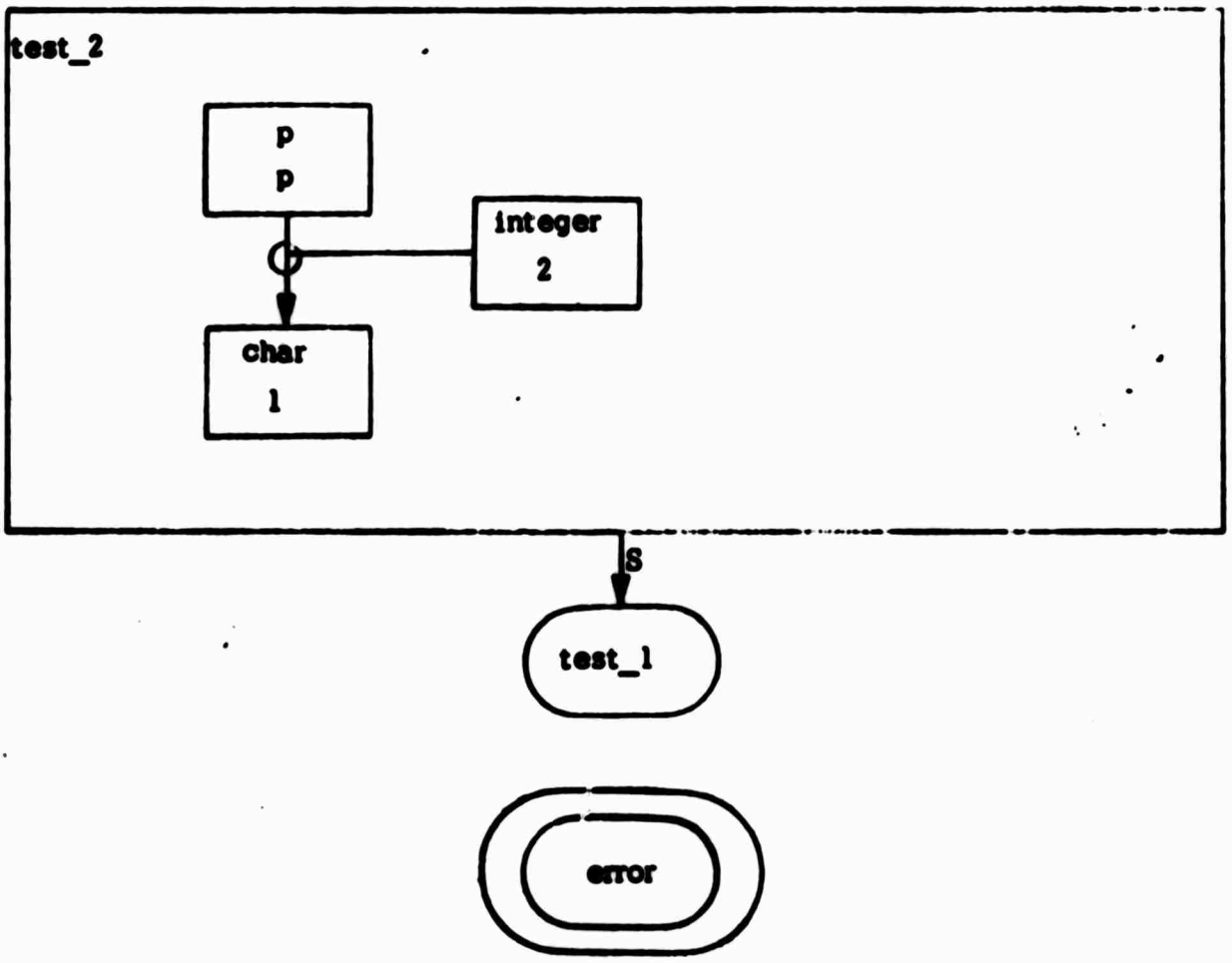
We have chosen to include listings of loader input to represent the program in two different forms. First the listing of 'msw3' is given which encodes the six rules using '-rule-' statements, etc. The second listing of 'msw2' represents exactly the same program (except for a name change), but

msw3


msw3
cell
x
cell
end
p
cell
x
read_function
builtin
link
integer
2
link
write_function
load
test
msw3


test
p
p
integer
2
char
0
test_1


test_1
p
p
char
0
char
1
char
char
F
integer
2
S
F
test_2
stop


test_2
p
p
integer
2
char
1
S
test_1
error

the entire loader input consists of loading the completely "desugared" form of rules as AMBIT/G data. We provide this comparison as a demonstration that there really is nothing more to a rule than its representation as data, and to emphasize the value of a loader which can accept rules. Originally, our modest goals of system design did not include the loading of rules. This very example was most effective in causing an extension of that design.

```
-page- slab 1 of msw3
-rule- r1 msw3
a1 cell x
a2 cell end
b1 type p
---
a1 next! a2
a1 value! b1
-endrule-
-rule- r2
a1 cell x
b3 builtin link
c1 integer 2
d1 link write_function
---
(a1,c1) read_function! b3
(a1, c1) :d1! b3
-endrule-
-rule- r3
node
---
() load node
-endrule-
-ruleref- r4 test
-links-
r1 success r2
r2 success r3
r3 success r4
-start- msw3
```

(Cont' on next page)

```
-page- first rule of slab 2 of msw3
-rule- test test
a1 p p
b1          char        0
i2          integer     2
---
a1          :i2!        b1
-endrule-
-ruleref- test_1 test_1
-links-
test success test_1

-page- second rule
-ruleref- ruleref test_2
-rule- test_1 test_1
pp p p
c0 char 0
c1 char 1
d
f char F
i2 integer 2
---
pp :i2? c0
pp :i2! c1
pp :i2 d
f char! d
-endrule-

-ruleref- stop stop

-links-
test_1 success ruleref
test_1 fail stop

-page- third rule
-rule- r1 test_2
a1 p p
i2 integer 2
b1 char 1
---
a1 :i2 b1
-endrule-
-ruleref- r2 test_1
-links-
r1 success r2
-start- error
```

```
-page- slab 1 of msw2
r1 rule msw2
r2 rule
r3 rule
r4 rule test
fc flag clear
ff flag frame
fm flag modify
ffix flag fixed
fdum flag dummy
le linkrep end
l1 linkrep
l2 linkrep
l3 linkrep
l4 linkrep
l5 linkrep
de diamond end
d1 diamond
d2 diamond
d3 diamond
d4 diamond
d5 diamond
d6 diamond
d7 diamond
d8 diamond
d9 diamond
d10 diamond
d11 diamond
na1 noderep
na2 noderep
na3 noderep
na4 noderep
na5 noderep
na6 noderep
na7 noderep
nlv noderep
nln noderep
nlrf noderep
nlwf noderep
nll noderep
lv link value
ln link next
read_function link read_function
write_function link write_function
load link load
a1 cell x
a2 cell end
a3 type p
a4 integer 2
a5 builtin link
a6 cell x
```

(Cont' on next page)

```
-links-
r1 success r2
r1 contents l1
r1 state fc
r2 success r3
r2 contents l3
r2 state fc
r3 success r4
r3 contents l5
r3 state fc
l1 next l2
l2 next le
l3 next l4
l4 next le
l5 next le
l1 org d1
l1 name n1n
l1 dest d2
l1 mode fm
l2 org d3
l2 name n1v
l2 dest d4
l2 mode fm
l3 org d5
l3 name n1rf
l3 dest d7
l3 mode fm
l4 org d8
l4 name n1wf
l4 dest d10
l4 mode fm
l5 org de
l5 name n11
l5 dest d11
l5 mode ff
d1 next de
d1 value na1
d2 next de
d2 value na2
d3 next de
d3 value na1
```

(Cont' on next page)

```
d4 next de
d4 value na3
d5 next d6
d5 value na6
d6 next de
d6 value na4
d7 next de
d7 value na5
d8 next d9
d8 value na6
d9 next de
d9 value na4
d10 next de
d10 value na5
d11 next de
d11 value na7
na1 rep a1
na1 variability ffix
na2 rep a2
na2 variability ffix
na3 rep a3
na3 variability ffix
na4 rep a4
na4 variability ffix
na5 rep a5
na5 variability ffix
na6 rep a6
na6 variability ffix
na7 variability fdum
nlv rep lv
nlv variability ffix
nln rep ln
nln variability ffix
nlrf rep read_function
nlrf variability ffix
nlwf rep write_function
nlwf variability ffix
nll rep load
nll variability ffix
-start- msw2
```

(Cont' on next page)

```
-page- slab 2 of msw2
r1 rule test
r2 rule test_1
fc flag clear
fm flag modify
ff flag fixed
lend linkrep end
l1 linkrep
dend diamond end
d1 diamond
d2 diamond
npp noderep
nc0 noderep
ni2 noderep
pp p p
c0 char 0
i2 integer 2
-links-
r1 success r2
r1 contents l1
r1 state fc
l1 next lend
l1 org d1
l1 name ni2
l1 dest d2
l1 mode fm
d1 next dend
d1 value npp
d2 next dend
d2 value nc0
npp variability ff
npp rep pp
nc0 variability ff
nc0 rep c0
ni2 variability ff
ni2 rep i2
```

(Cont' on next page)

```
-page- second rule
rule rule test_1
next_rule rule test_2
stop rule stop
clear flag clear
frame flag frame
test flag test
modify flag modify
fixed flag fixed
dummy flag dummy
lend linkrep end
l1 linkrep
l2 linkrep
l3 linkrep
l4 linkrep
dend diamond end
d1 diamond
d2 diamond
d3 diamond
d4 diamond
d5 diamond
d6 diamond
d7 diamond
d8 diamond
np_p noderep
nchar_0 noderep
nchar_1 noderep
nchar_F noderep
ndummy noderep
ninteger_2 noderep
nchar noderep
p_p p p
char_0 char 0
char_1 char 1
char_F char F
integer_2 integer 2
char link char
```

(Cont' on next page)

```
-links-
rule success next_rule
rule fail stop
rule contents l1
rule state clear
l1 next l2
l2 next l3
l3 next l4
l4 next lend
l1 org d1
l1 name ninteger_2
l1 dest d2
l1 mode test
l2 org d3
l2 name ninteger_2
l2 dest d4
l2 mode modify
l3 org d5
l3 name ninteger_2
l3 dest d6
l3 mode frame
l4 org d7
l4 name nchar
l4 dest d8
l4 mode modify
d1 next dend
d1 value np_p
d2 next dend
d2 value nchar_0
d3 next dend
d3 value np_p
d4 next dend
d4 value nchar_1
d5 next dend
d5 value np_p
d6 next dend
d6 value ndummy
d7 next dend
d7 value nchar_F
d8 next dend
d8 value ndummy
np_p variability fixed
np_p rep p_p
nchar_0 variability fixed
nchar_0 rep char_0
nchar_1 variability fixed
nchar_1 rep char_1
nchar_F variability fixed
nchar_F rep char_F
ndummy variability dummy
ninteger_2 variability fixed
ninteger_2 rep integer_2
nchar variability fixed
nchar rep char
```

(Cont' on next page)

```
-page- third rule
r1 rule test_2
r2 rule test_1
fc flag clear
ft flag test
ff flag fixed
lend linkrep end
l1 linkrep
dend diamond end
d1 diamond
d2 diamond
npp noderep
nc1 noderep
ni2 noderep
pp p p
c1 char 1
i2 integer 2
-links-
r1 success r2
r1 contents l1
r1 state fc
l1 next lend
l1 org d1
l1 name ni2
l1 dest d2
l1 mode ft
d1 next dend
d1 value npp
d2 next dend
d2 value nc1
npp variability ff
npp rep pp
nc1 variability ff
nc1 rep c1
ni2 variability ff
ni2 rep i2
-start- error
```

FUNCTION CALLING

During system development the program 'msw5' was designed as a test of the function calling mechanism of the AMBIT/G interpreter. It is included here to show the generality of function definition and calling.

Page '5-2' includes three definitions of what the interpreter should do in processing a link named 'eq_any'. If no tails are given, its definition is 'builtin type'. If any two tails are given, the user function beginning at rule 'eq_any_2' should be called. Otherwise, for any other number of tails the user function beginning at rule 'eq_any_not_2' should be called. Page '5-3' includes three such calls. The call with five arguments is a tested read call. The call with no tails causes an attempt to call the builtin 'type' with no tails, and that is detected as an error. The following listing of a run demonstrates this. Note that this run took approximately 4 minutes of CPU time.

```
r 2355  .195  5+1

→ ambitg msw5
AMBIT/G


AMBIT/G Error:      The interpreter has detected a wrong number of
          tails or heads on a read-call on the builtin "type".
This error occurred while interpreting the rule "rule &1247".
The interpreter was processing the link  "()   eq_any   (  )".

r 0  241.534  28+473
```

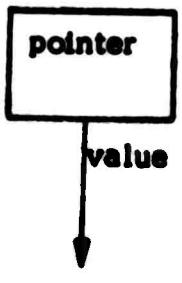
Page '5-4' contains the definition of the function for testing the equality of its two arguments. This has already been discussed in the 'reverse3' example.

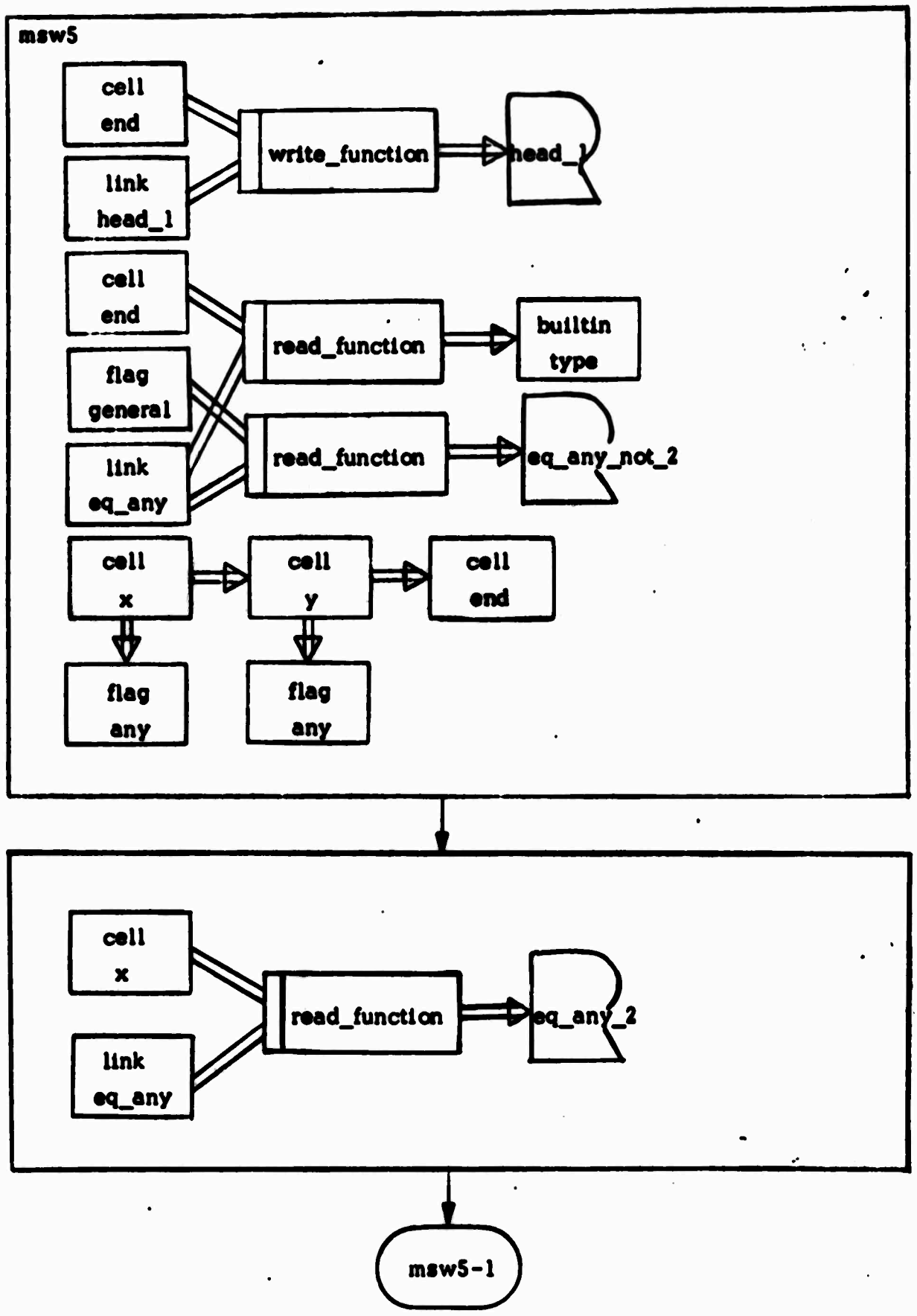
Page '5-5' contains the definition of the function for testing whether its first argument is equal to any of its others. A walk is made down the list of 'pipe's representing the tail arguments. Note the use of the write function 'head_1'; it makes the writing of a function easier and its later reading more

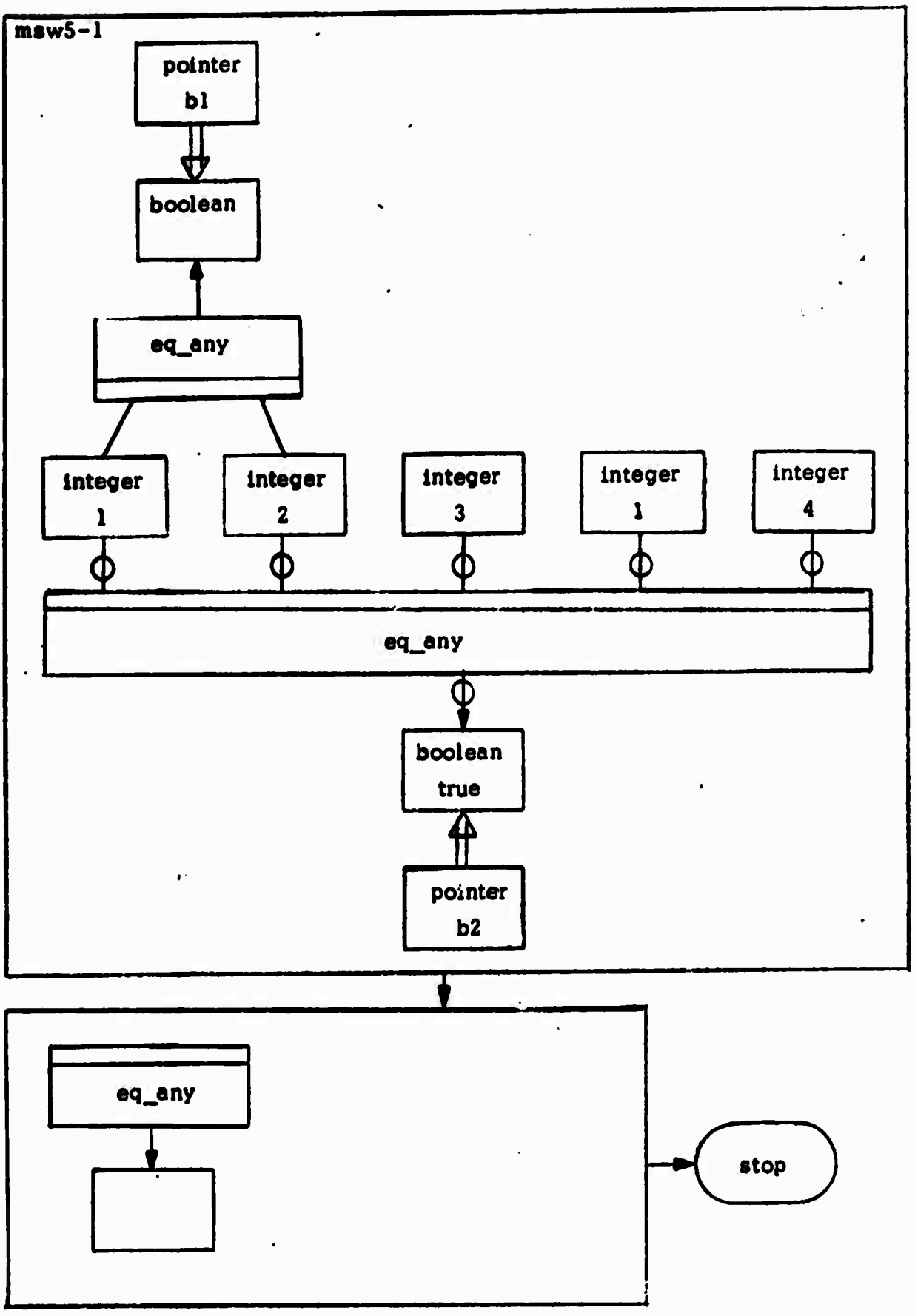
clear. It, of course, is used to return a result to the caller.

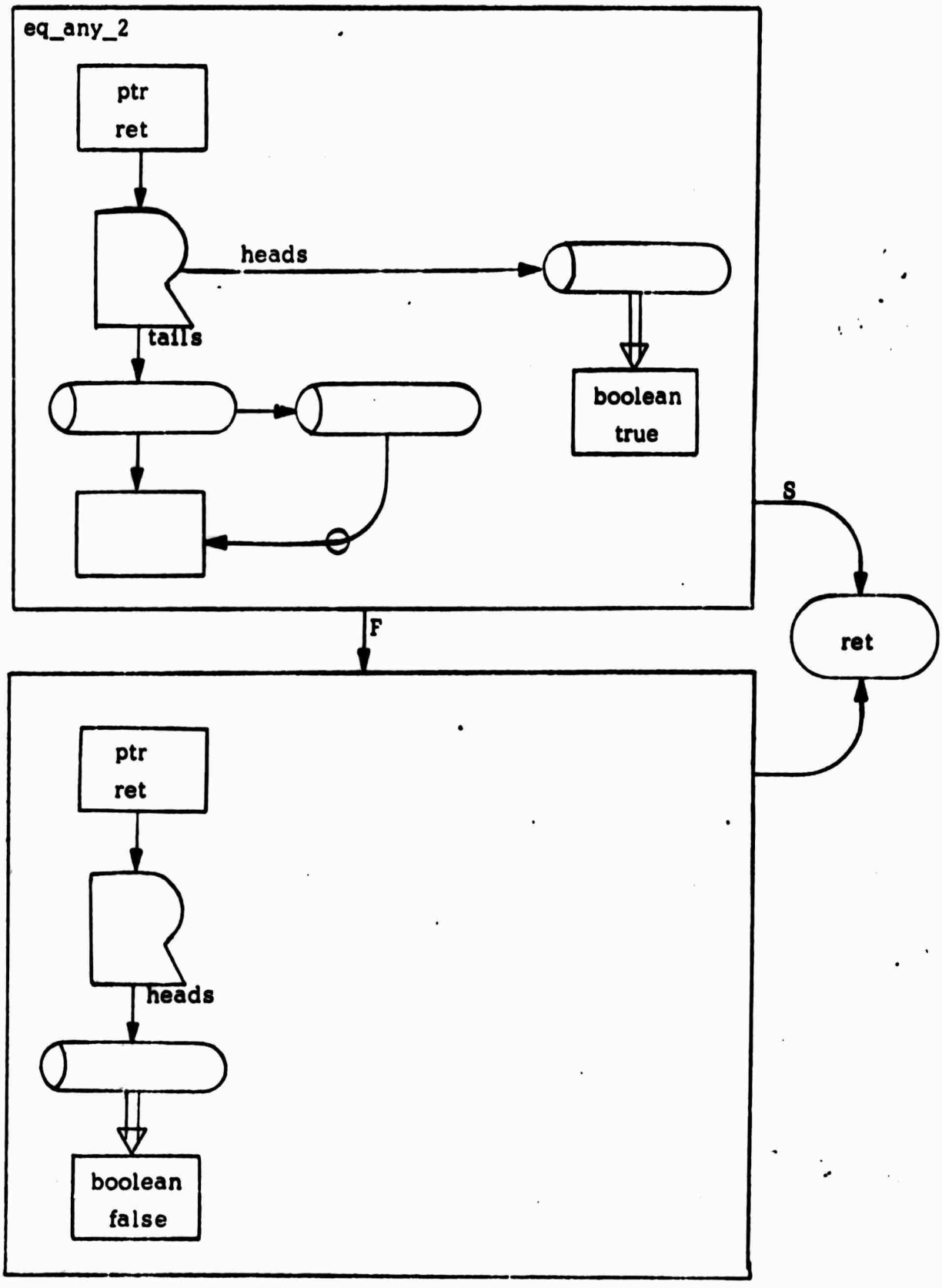
The 'head_1' function is given on page '5-6'. Note that it must walk up the call stack one extra level by a 'saveret' link so that it can alter the result of the function which called it. Similar functions can be written for obtaining head and tail arguments.

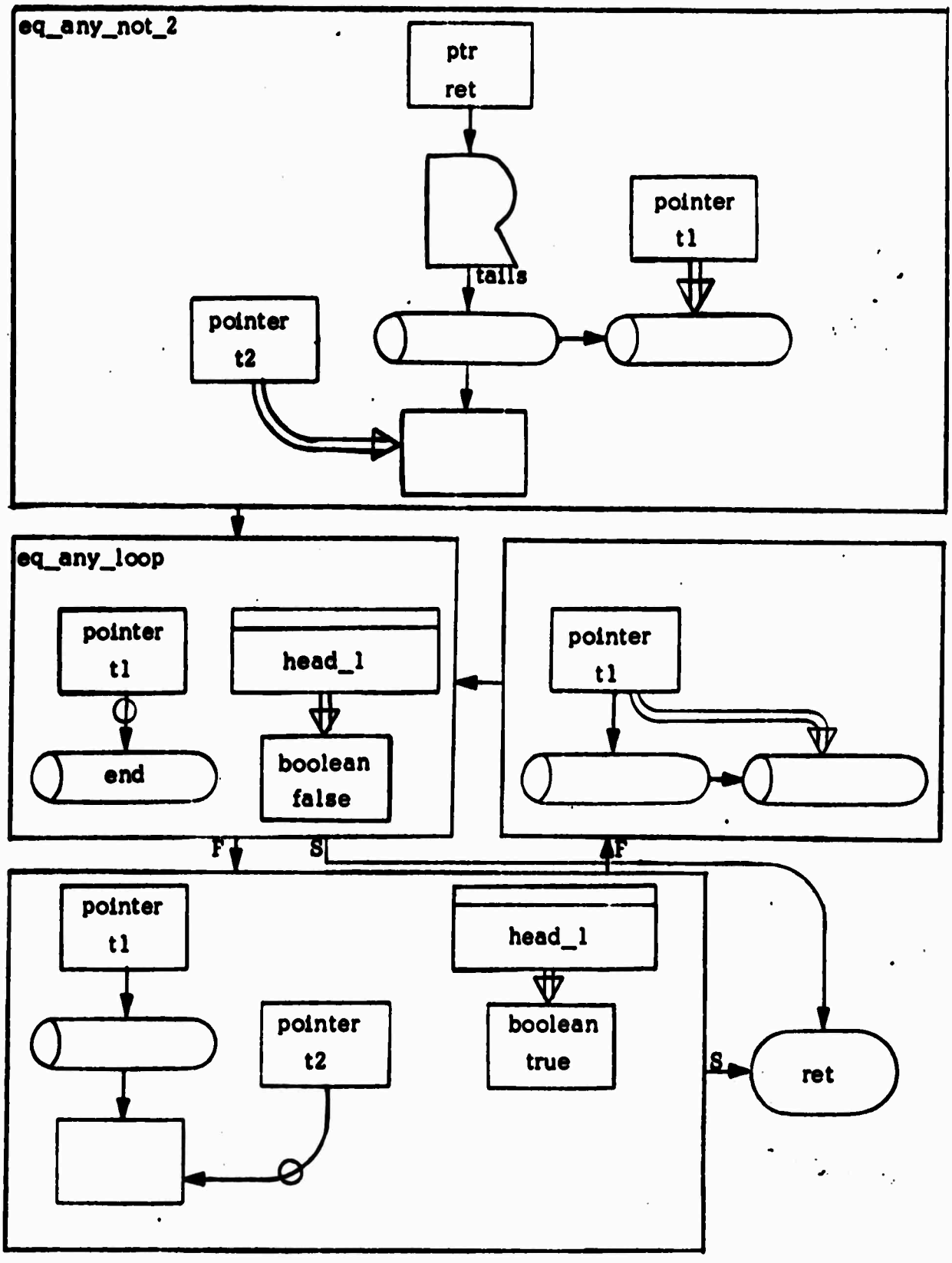
msw5

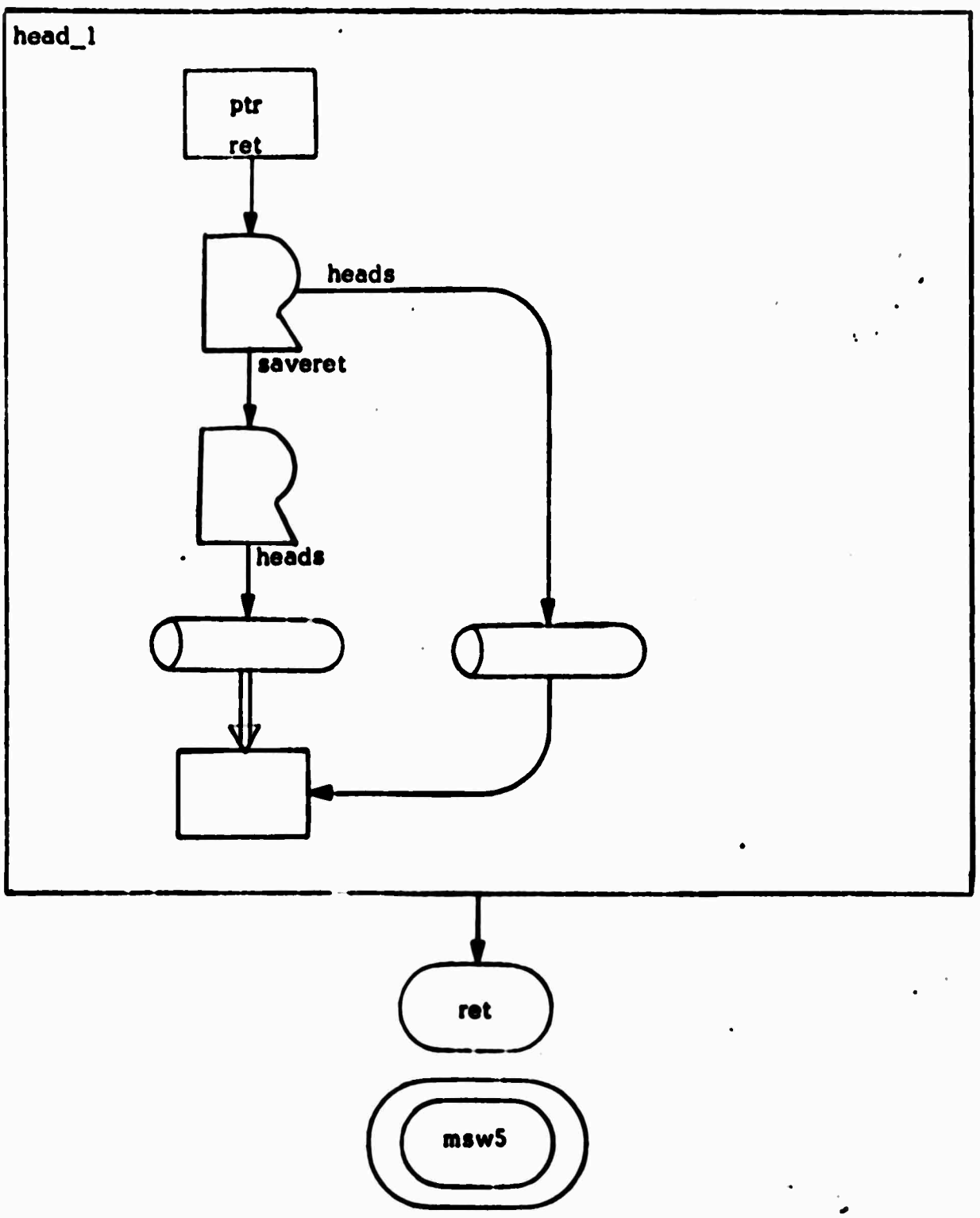



msw5
cell
end
link
head_1
write_function
head_1
cell
end
flag
general
link
eq_any
read_function
builtin
type
read_function
eq_any_not_2
cell
x
cell
y
cell
end
flag
any
flag
any
cell
x
link
eq_any
read_function
eq_any_2
msw5-1


msw5-1
pointer
b1
boolean
eq_any
integer
1
integer
2
integer
3
integer
1
integer
4
eq_any
boolean
true
pointer
b2
eq_any
stop


eq_any_2
ptr
ret
heads
tails
boolean
true
S
F
ret
ptr
ret
heads
boolean
false


eq_any_not_2
ptr
ret
tails
pointer
t1
pointer
t2
eq_any_loop
pointer
t1
head_1
end
boolean
false
pointer
t1
F
S
F
pointer
t1
head_1
pointer
t2
boolean
true
S
ret


head_1
ptr
ret
heads
saveret
heads
ret
msw5

A LISP GARBAGE COLLECTOR

We present in this section the program 'lispgc' which represents an algorithm used for garbage collection in some implementations of LISP. The program was used as an example in a paper by Carlos Christensen presented at the Symposium on Interactive Systems for Experimental Applied Mathematics in August, 1967.*

Since then, the AMBIT/G language has undergone revision and thus the listings included here differ in some details with those in the original paper.

This example makes use of user-defined characteristic shapes. A '¬=' function is defined for determining whether its two tail arguments are not equal.

'SQUARE's are used to represent the forks of a binary tree; the branches are represented by the 'left-son' and 'right-son' links which emerge from a 'SQUARE', and the leaves are 'CIRCLE's. For the purpose of garbage collection the 'SQUARE's must be organized in a single sequence (at the same time they are being used in the tree), and the 'down' link of the 'SQUARE's is used for this purpose. Links of the 'DIAMOND' and 'QUADRANT' nodes are used to keep track of scans and walks through the data. 'left' and 'right' links of 'SQUARE's are used for temporary indicators for the garbage collection algorithm.

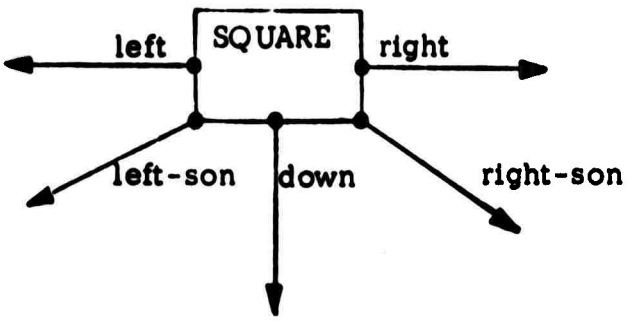
Page 'A' presents the types of nodes and any defined links. Page 'B' presents all nodes used in the program (this is unnecessary) and the constant links of the data.

Page 'C' can be considered as input to the program; it represents a free list and a particular tree of LISP data which includes some garbage.

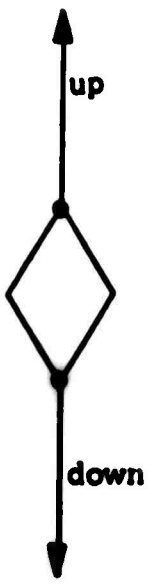
---

* This paper, entitled "An Example of the Manipulation of Directed Graphs in the AMBIT/G Programming Lanugage", is published in Interactive Systems for Experimental Applied Mathematics edited by M. Klerer and J. Reinfelds. This book is published by Academic Press, New York (1968).

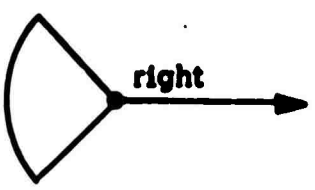
lispgc



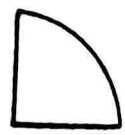
DIAMOND:



QUADRANT:

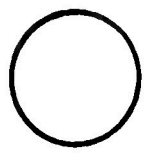


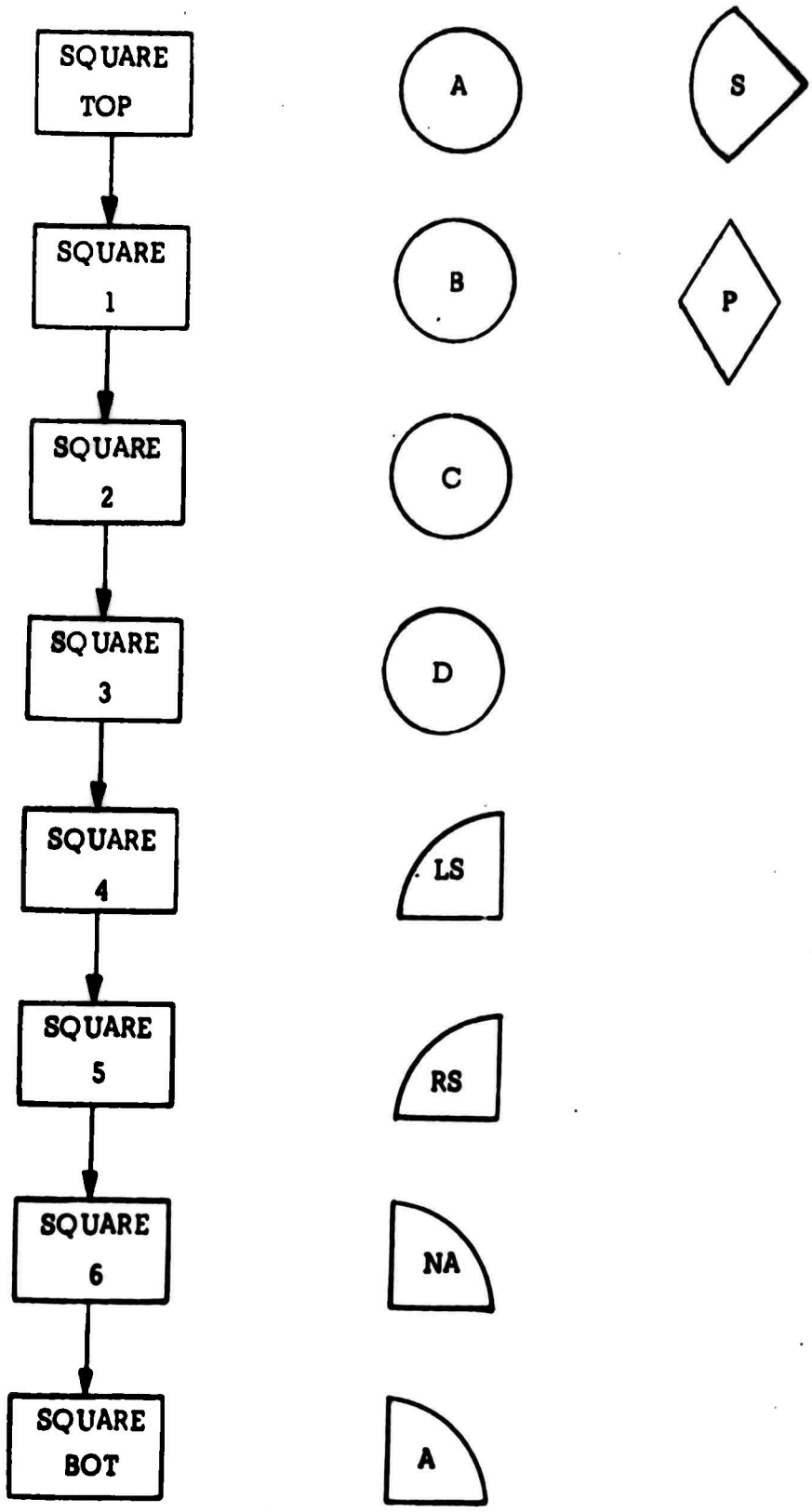
RIGHT-QUAD :

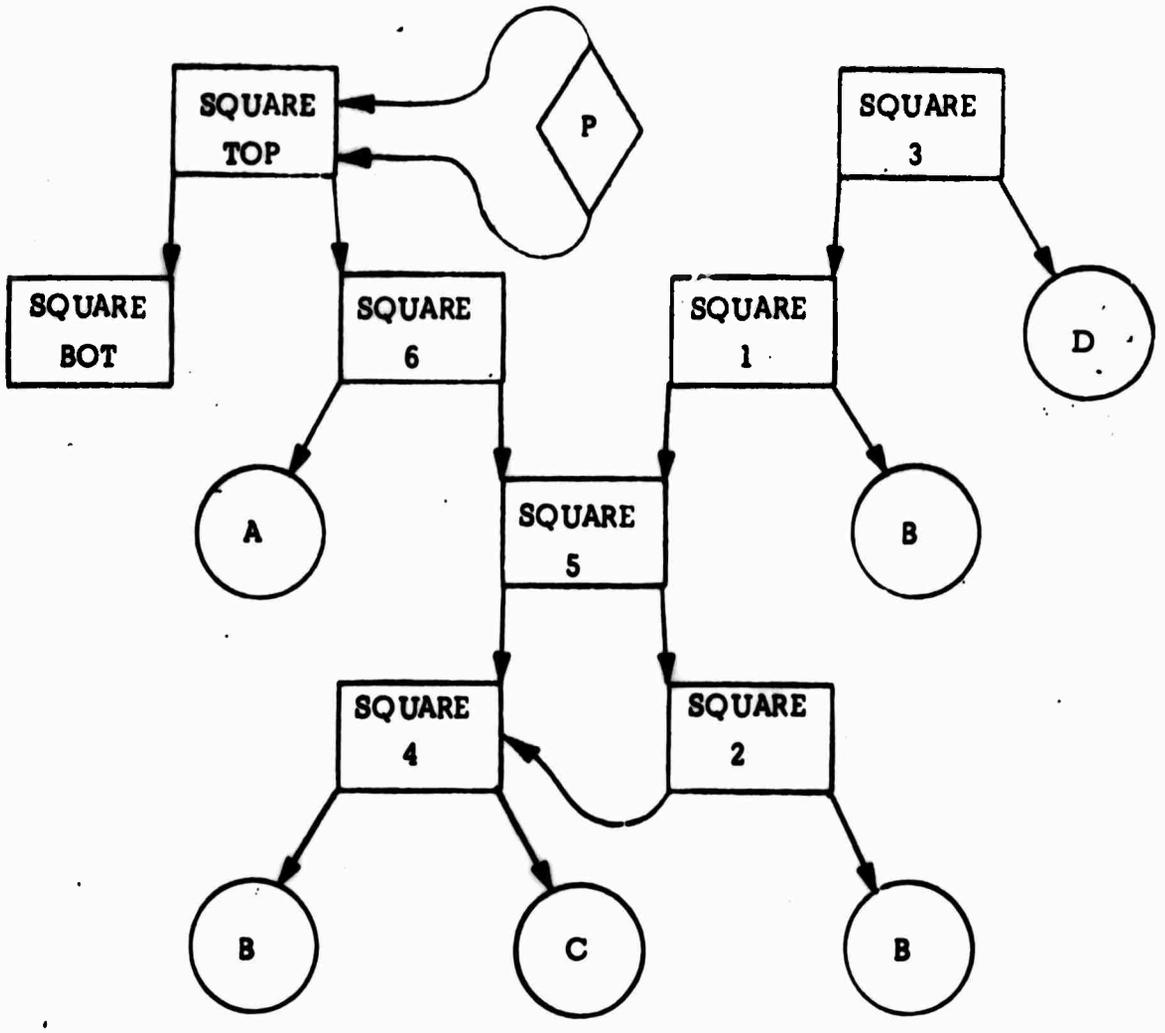


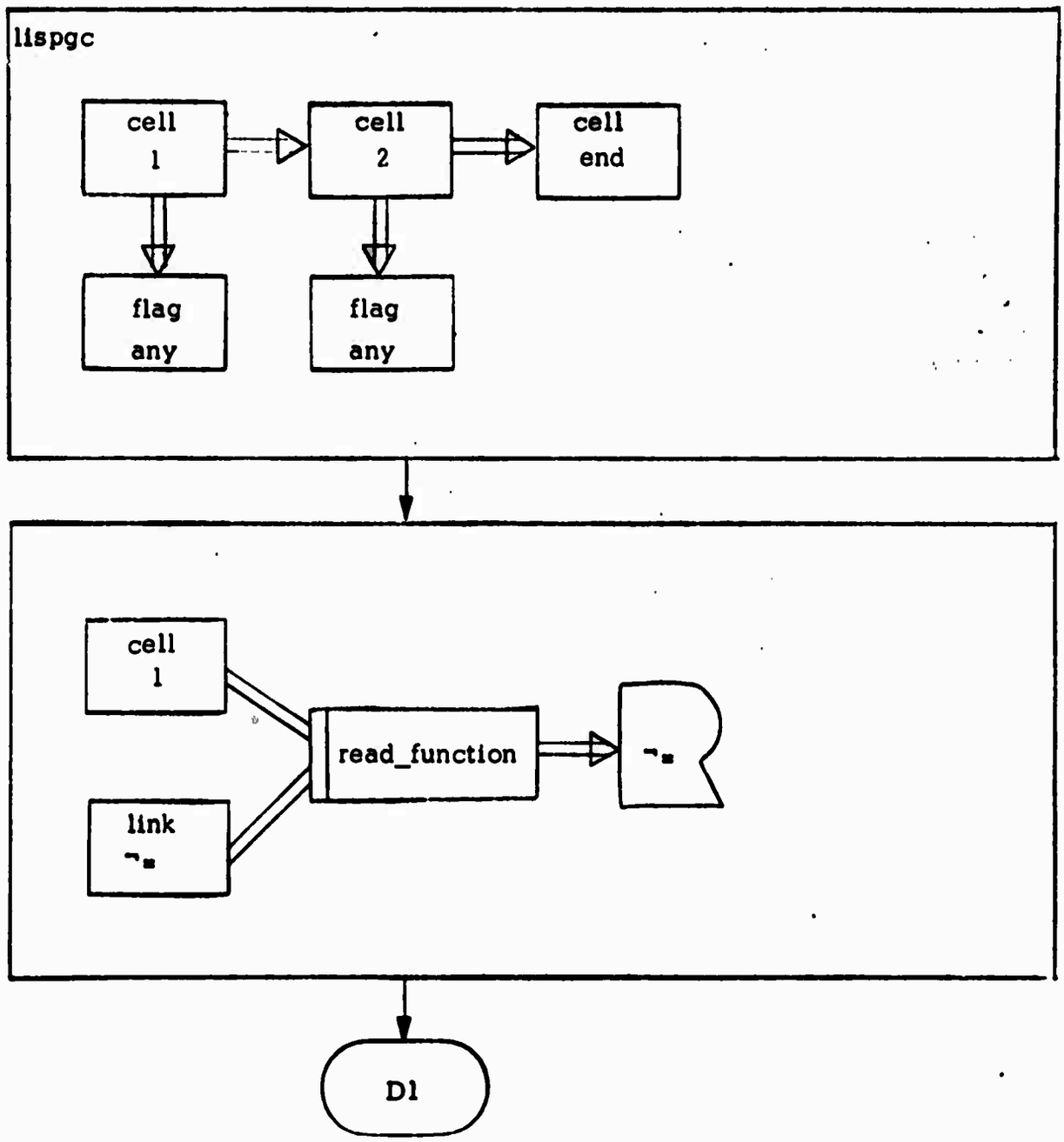
LEFT-QUAD :

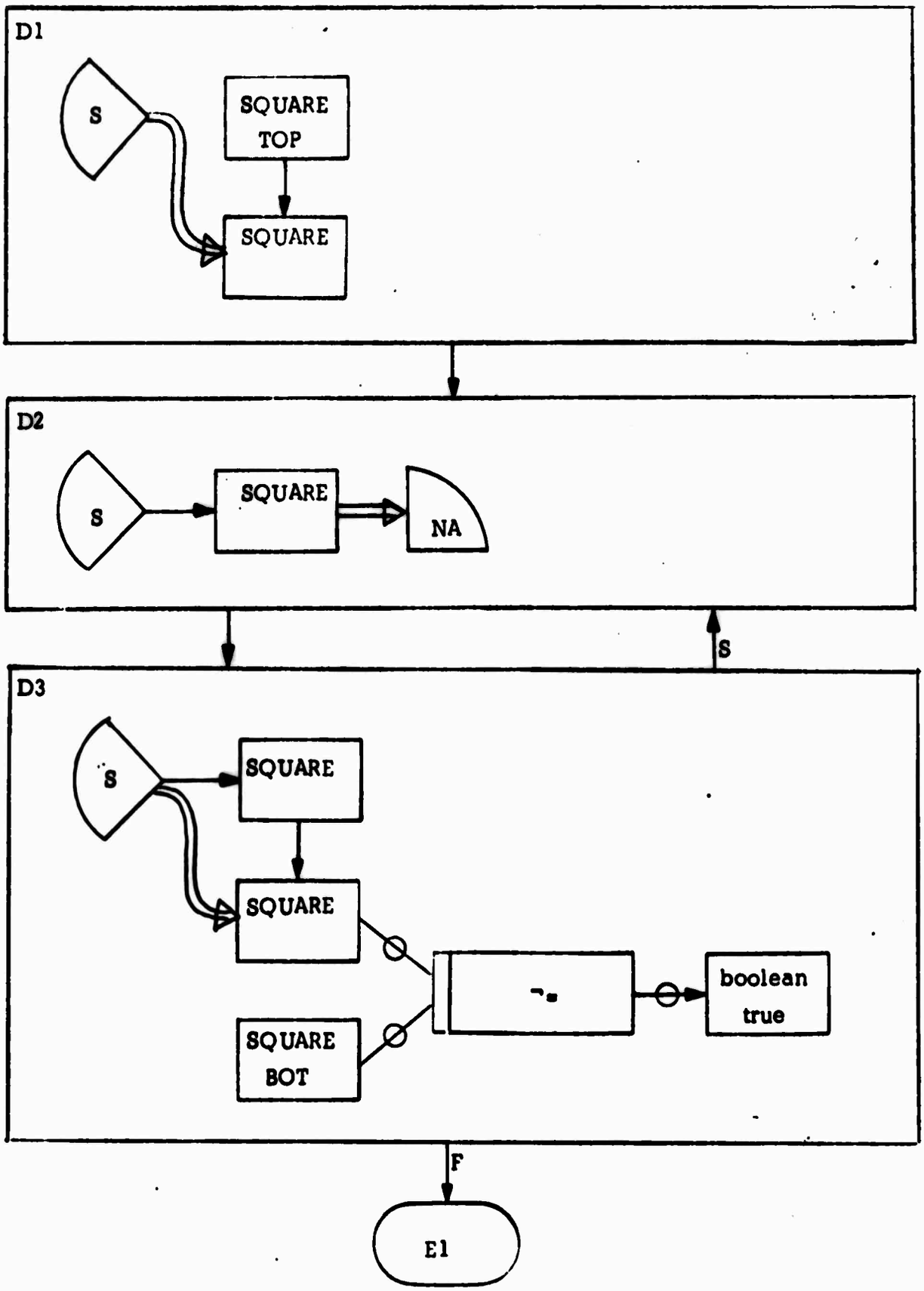
CIRCLE :

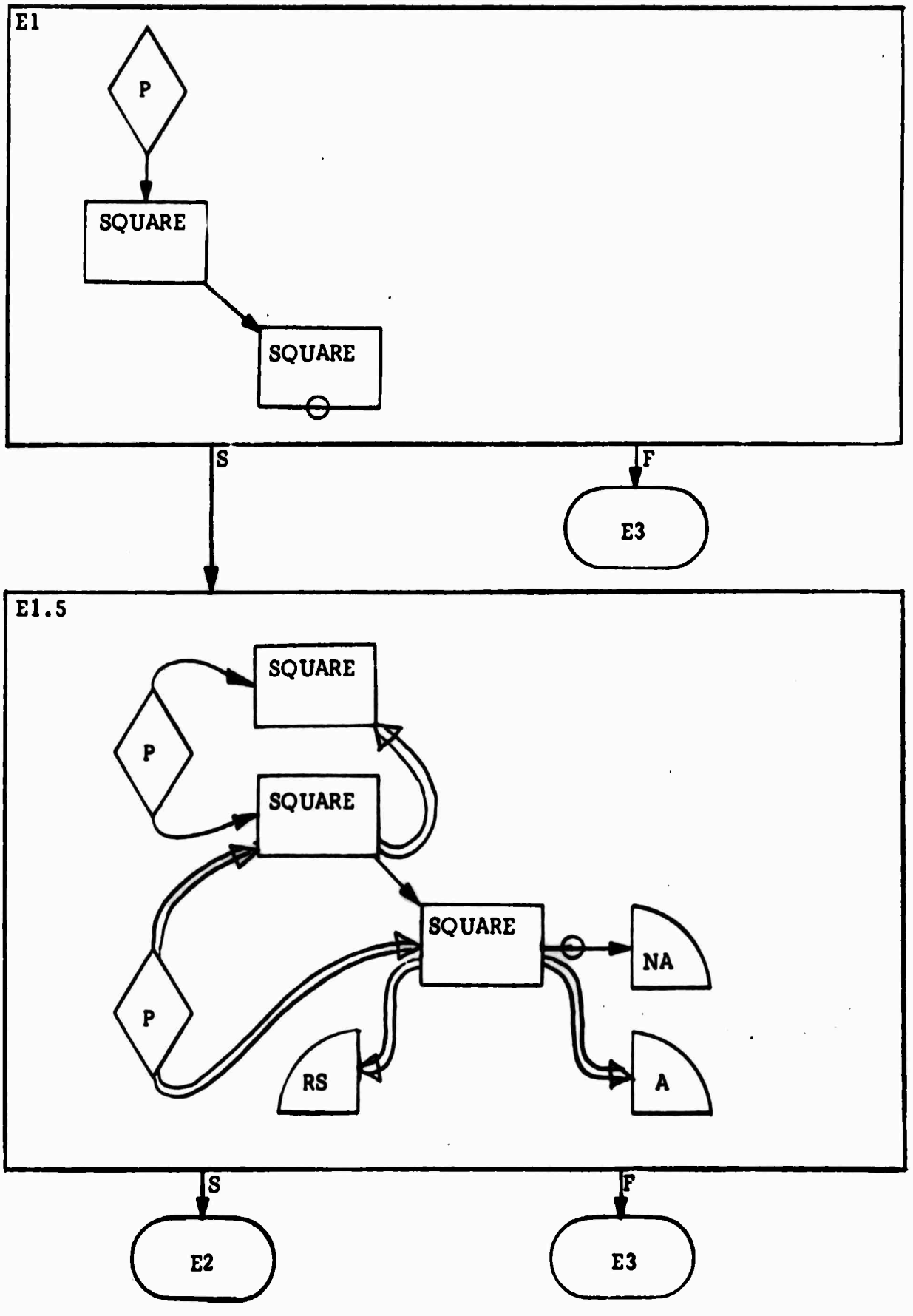


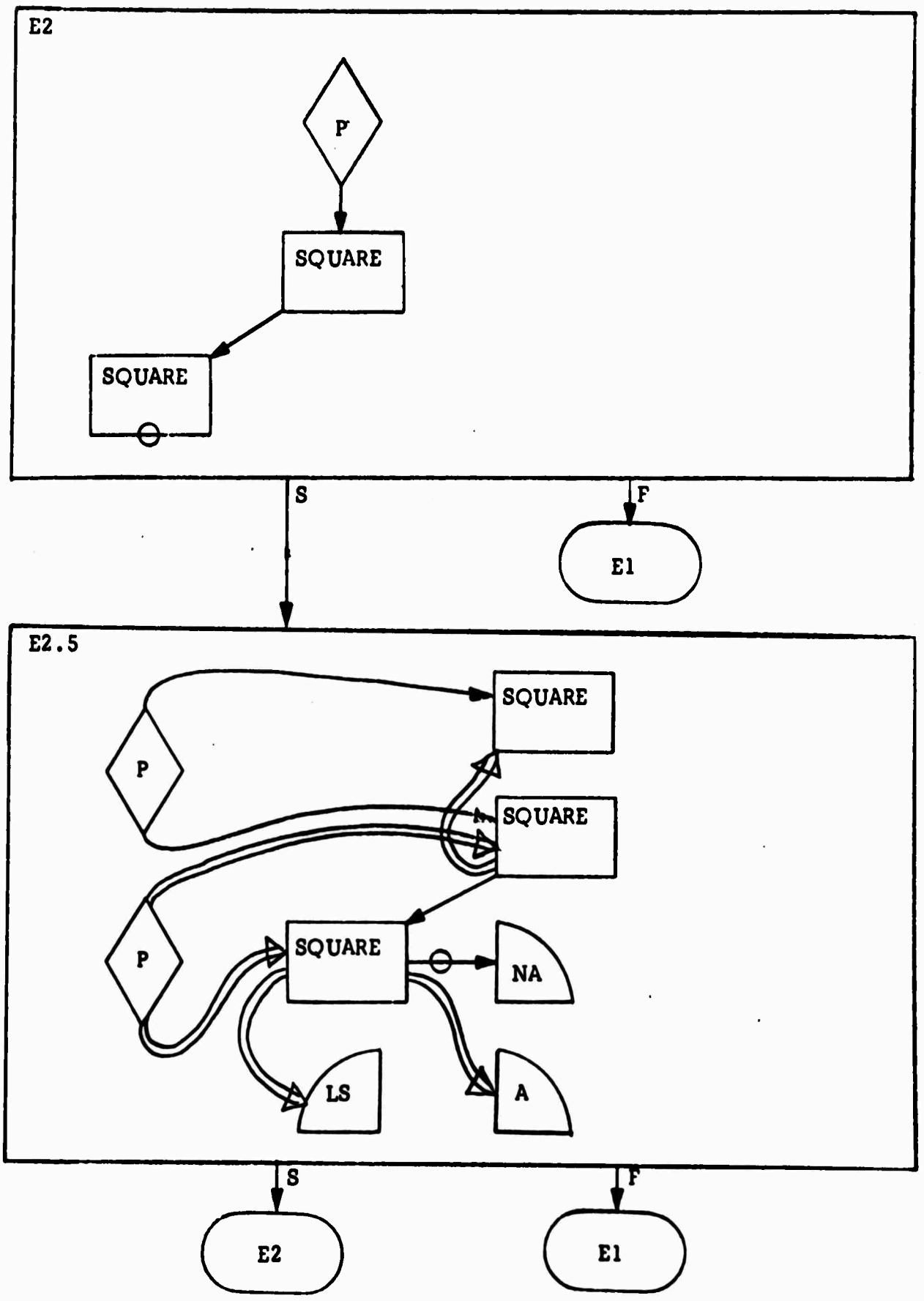

SQUARE TOP
SQUARE 1
SQUARE 2
SQUARE 3
SQUARE 4
SQUARE 5
SQUARE 6
SQUARE BOT
A
B
C
D
LS
RS
NA
A
S
P

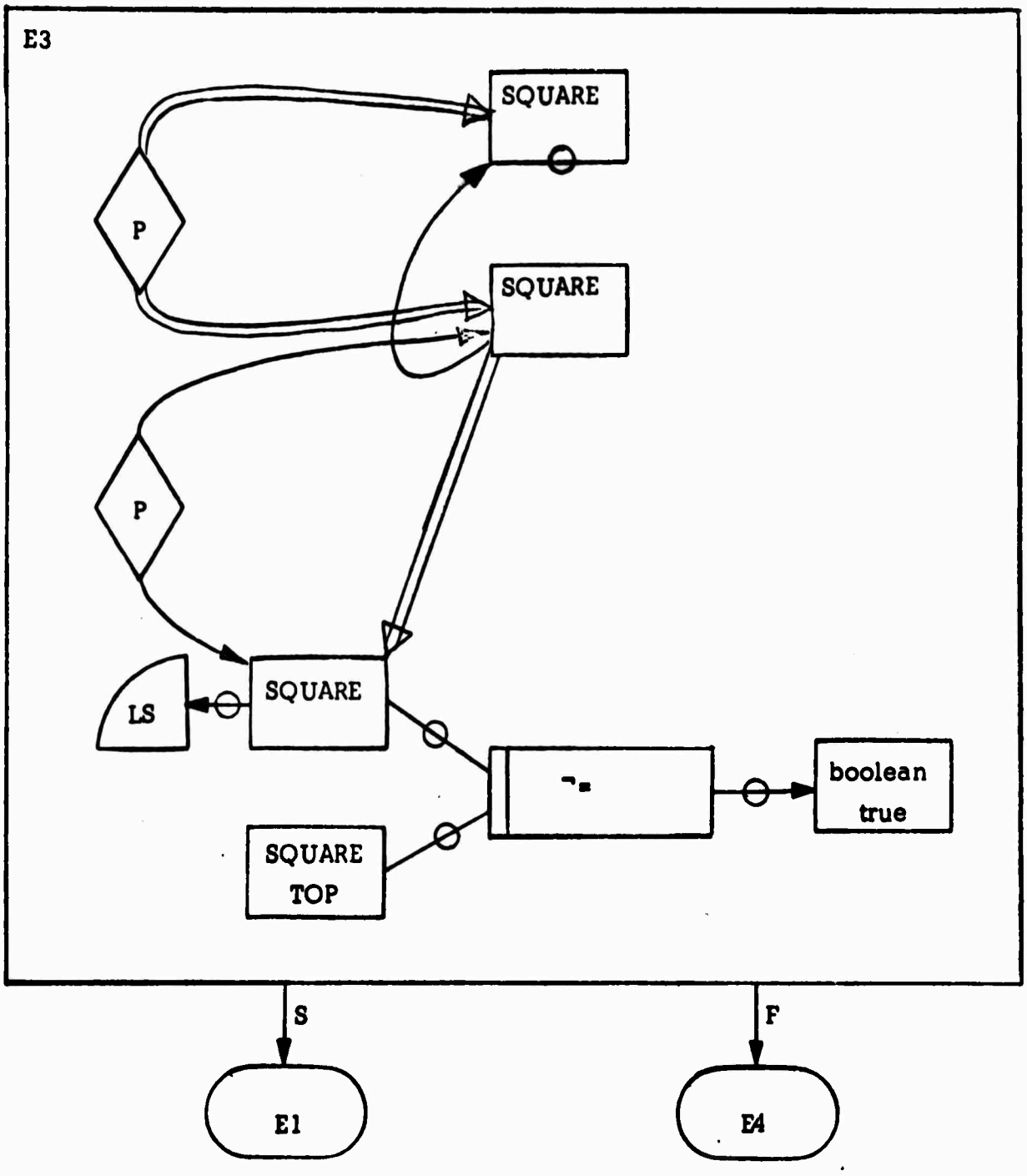

SQUARE
TOP
P
SQUARE
3
SQUARE
BOT
SQUARE
6
SQUARE
1
D
A
SQUARE
5
B
SQUARE
4
SQUARE
2
B
C
B

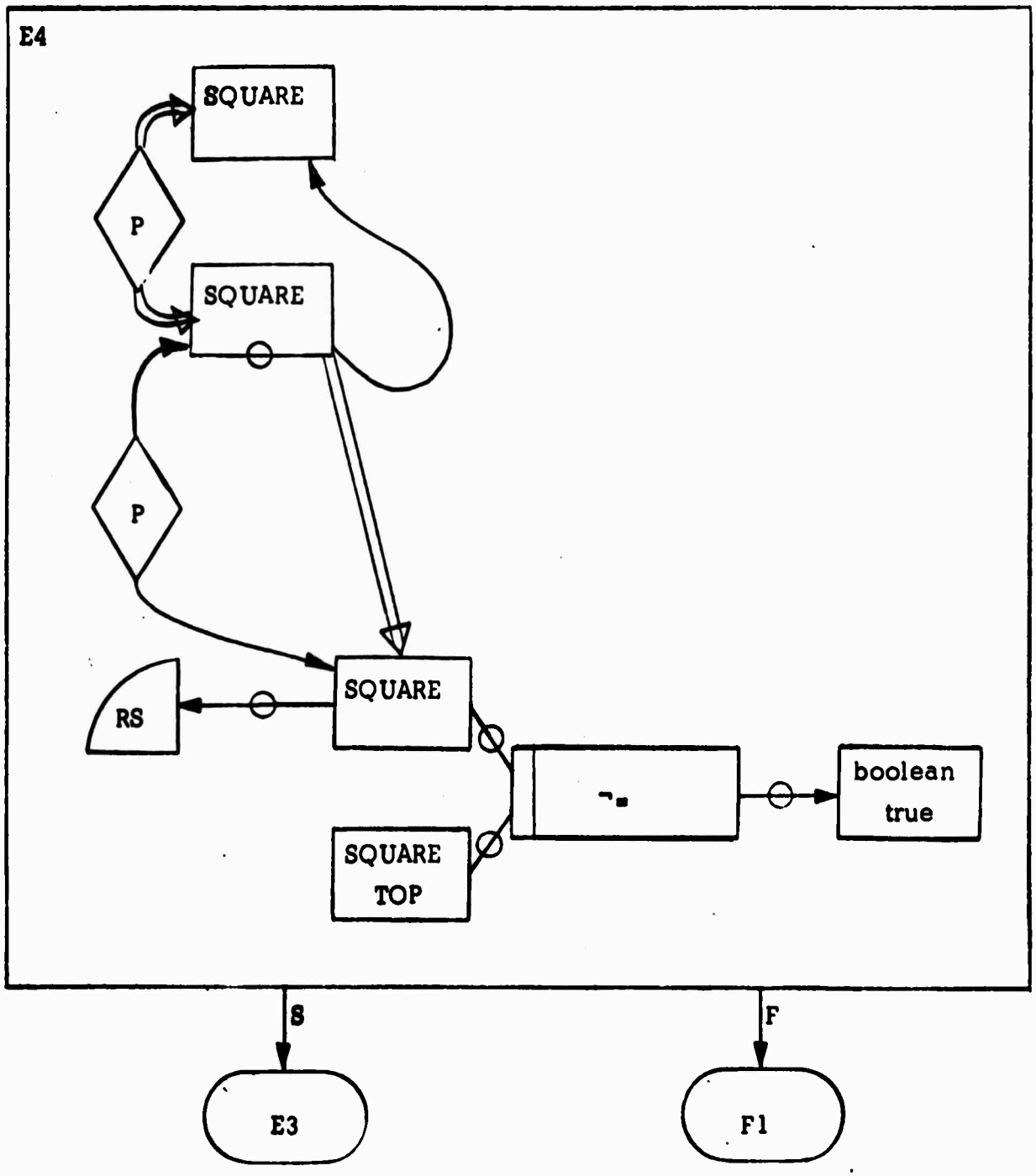

lispgc
cell 1
cell 2
cell end
flag any
flag any
cell 1
link ¬=
read_function
¬=
D1

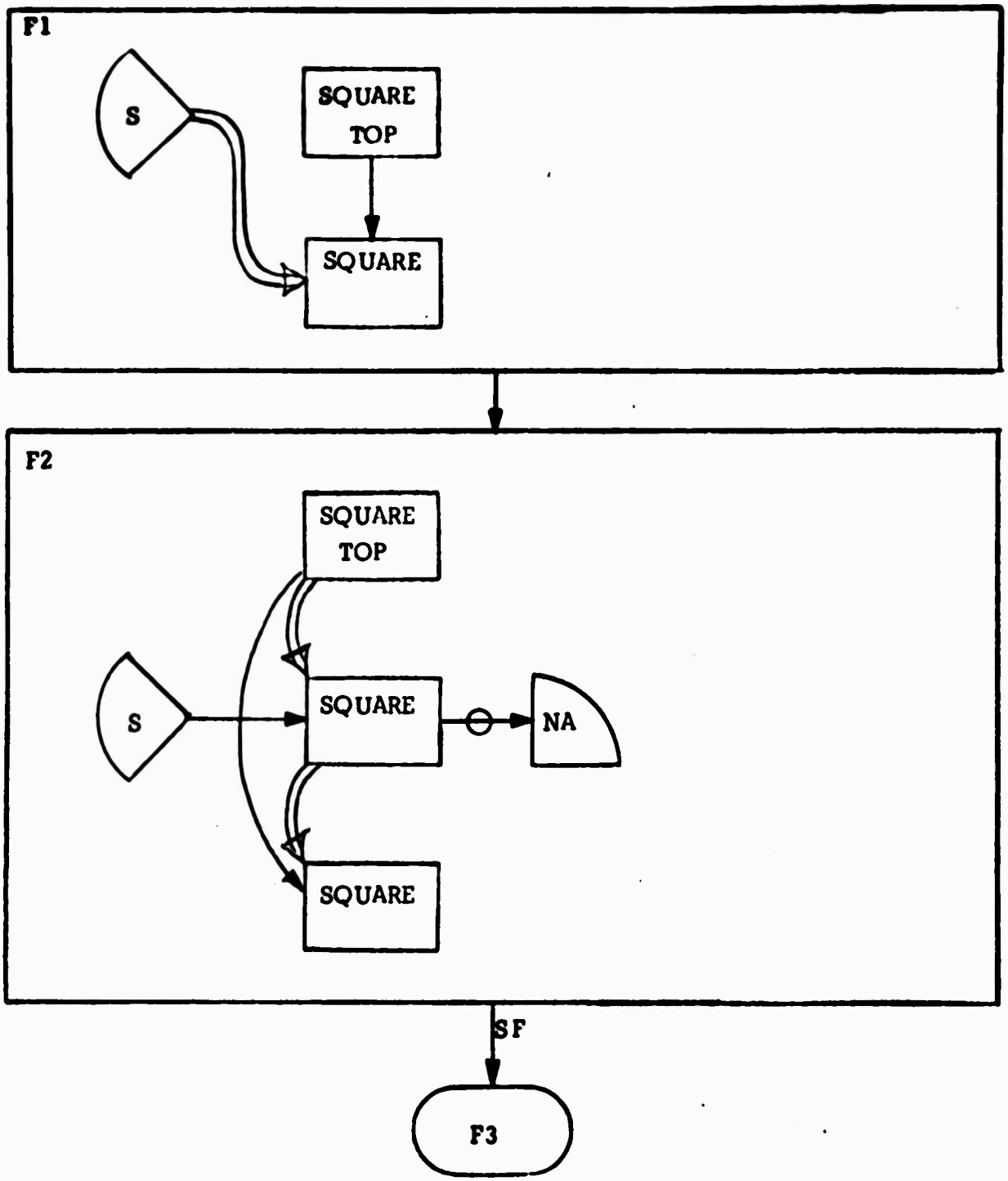

D1
S
SQUARE TOP
SQUARE
D2
S
SQUARE
NA
S
D3
S
SQUARE
SQUARE
SQUARE BOT
boolean true
F
E1

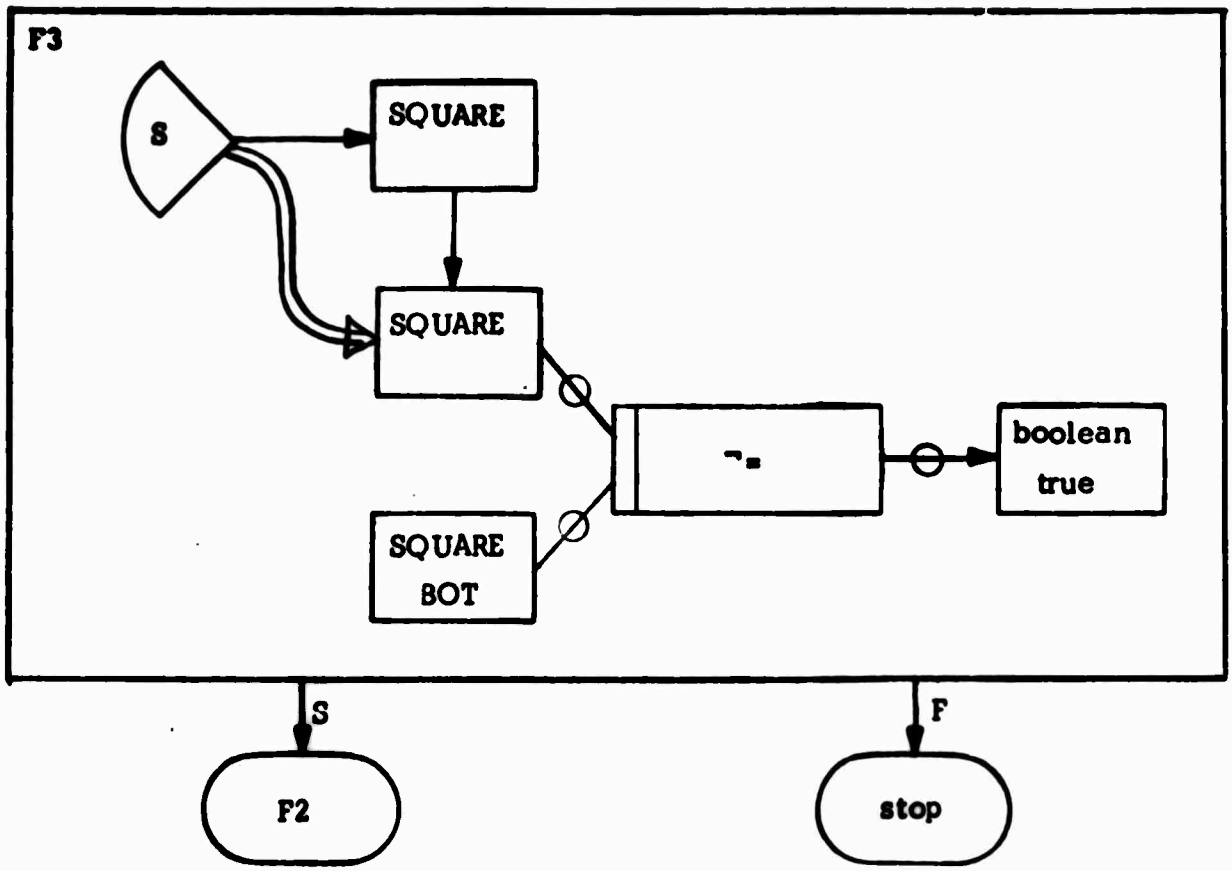

E1
P
SQUARE
SQUARE
S
F
E3
E1.5
SQUARE
P
SQUARE
SQUARE
NA
P
RS
A
S
F
E2
E3

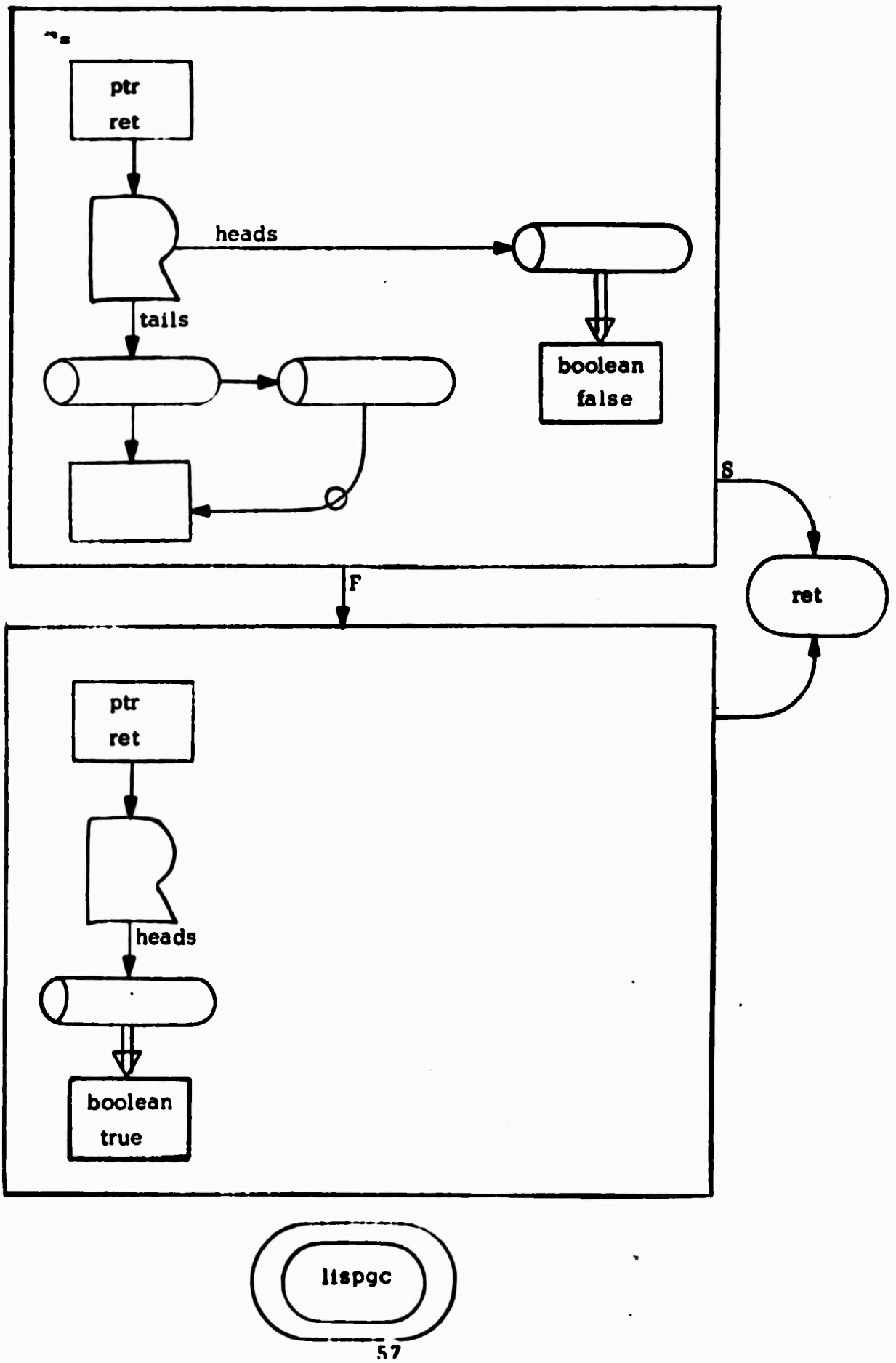

E2
P
SQUARE
SQUARE
S
F
E1
E2.5
P
SQUARE
SQUARE
P
SQUARE
NA
LS
A
S
F
E2
E1


E3
SQUARE
P
SQUARE
P
SQUARE
LS
¬=
boolean
true
SQUARE
TOP
S
F
E1
E4


E4
SQUARE
P
SQUARE
P
SQUARE
RS
¬
boolean
true
SQUARE
TOP
S
F
E3
F1


F1
S
SQUARE
TOP
SQUARE
F2
SQUARE
TOP
S
SQUARE
NA
SQUARE
SF
F3


F3
S
SQUARE
SQUARE
SQUARE
BOT
¬=
boolean
true
S
F
F2
stop


ptr
ret
heads
tails
boolean
false
S
ret
F
ptr
ret
heads
boolean
true
lispgc

Free 'SQUARE's, if there were any, would be chained by means of 'left-son' links between 'SQUARE TOP' and 'SQUARE BOT' ; but since the left son of 'SQUARE TOP' is 'SQUARE BOT', the free list is empty. The right son of 'SQUARE TOP' is the root of the tree. Note that 'SQUARE 1' and 'SQUARE 3' are the garbage, and it is the object of this program to enter such nodes into the free list.

Note that the given tree is re-entrant since 'SQUARE 4' is the left son of both 'SQUARE 5' and 'SQUARE 2'. The algorithm is designed for any form of re-entrant tree, including those with cycles.

The program proper begins by marking all 'SQUARE's "not accessible" in the rules on page 'D'. This marking refers to accessibility from the root of the tree by means of 'left-son' and 'right-son' links; it is performed tentatively, subject to later correction.

Page 'E' has six rules responsible for marking accessible 'SQUARE's "accessible". This part of the program begins at the root of the tree and walks the tree selecting every 'SQUARE' which is accessible from the root 'SQUARE' by way of 'left-son' and 'right-son' links. Each 'SQUARE' thus selected is marked "accessible", thus overriding the prior tentative setting.

A tree walk is a bit more complicated than a sequential scan. When a particular 'SQUARE' is selected, it is not possible to walk to both of its sons "simultaneously"; rather the walk must proceed to one of the sons (say the left son) and somehow provide to return later to walk to the other son (the right son).

This process is sometimes organized around a pushdown stack which is used to record those sons for whom selection has been deferred. However, this pushdown stack requires an amount of memory which depends on the size and shape of the tree being walked. Thus the use of a pushdown stack is not appropriate for a garbage collection algorithm which is, after all, invoked because available memory has been exhausted.

The method used on Page 'E' is a different one. It is based on a back-tracking technique which was invented by Peter Deutsch. As the walk moves down the tree, links are bent backward so that a link which normally points to the son of a 'SQUARE' is caused to point to the father of that 'SQUARE'. Thus it is possible to walk down the tree until a 'CIRCLE' is encountered, back up to a new downward path, walk down that path, and so on until the entire tree has been walked.

The 'up' and 'down' links of 'DIAMOND P' are used to control the walk. It is convenient to refer to the two 'SQUARE's pointed to by these links as the selected father and the selected son. Since 'SQUARE TOP' does not have a father, it is assumed to be its own father, and the algorithm begins and ends with 'SQUARE TOP' as both the selected son and selected father. After each step of the walk the links which are bent back are exactly those which would normally trace the line of descent from 'SQUARE TOP' to the current selected son. Two links are required to control the walk because the tree structure is always broken between the selected father and the selected son.

The rules on Page 'E' represent the four steps used in the tree walk: father to right son (rules 'E1' and 'E1.5'), father to left son (rules 'E2' and 'E2.5'), left son to father (rule 'E3'), and right son to father (rule 'E4'). The father-to-son steps must record whether the new selected son is a right son (by 'RIGHT-QUAD RS') or a left son (by 'LEFT-QUAD LS') by setting the left link of the 'SQUARE'. This is necessary because this information is otherwise lost when the son link is bent back by the execution of the step.

The father-to-son steps each fail under either of two conditions. First, if the new selected son would be a 'CIRCLE', then the step is not taken because a leaf of the tree has been reached. Second, if the new selected son would be one which is marked "accessible", then the step is not taken because that son is a root of a subtree which has already been walked by way of some re-entrant link.

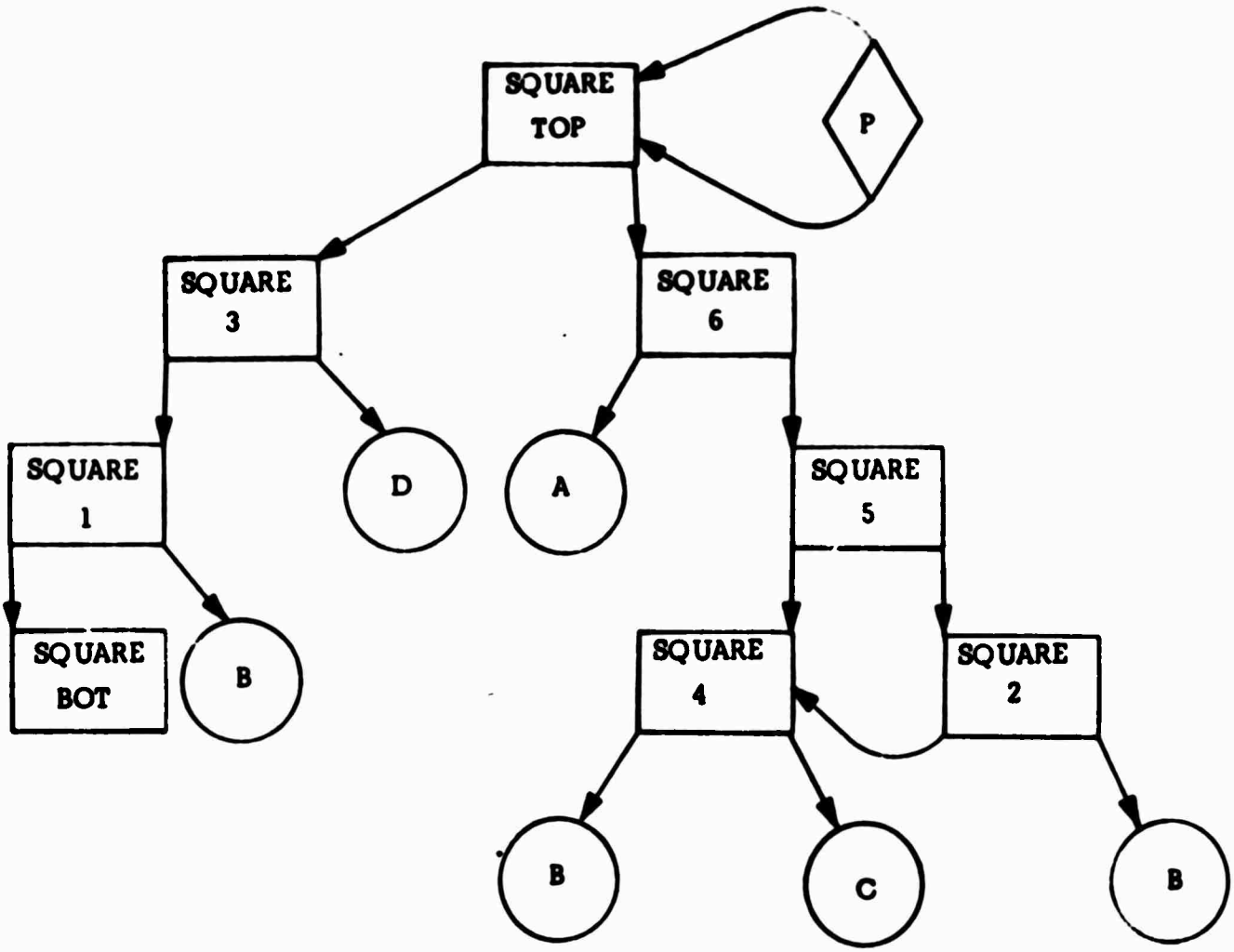
The son-to-father steps each fail under either of two conditions. First, if the selected son is not the correct son (left for rule 'E3', right for rule 'E4'), then the step is not taken. Second, if the selected son is 'SQUARE TOP', then the step is not taken because the walk is complete.

For the reader who wishes to experiment with a desk execution of the rules on Page 'E', the following trace is included:

E1-S,	E1.5-S,	E2-F,
E1-S,	E1.5-S,	E2-S,
E2.5-S,	E2-F,	E1-F,
E3-S,	E1-S,	E1.5-S,
E2-S,	E2.5-F,	E1-F,
E3-F,	E4-S,	E3-F,
E4-S,	E3-F,	E4-S,
E3-F,	E4-F	

This trace is read as "rule 'E1' is executed and the 'S' exit ('success') is taken, rule 'E1.5' is then executed and the 'S' exit is taken, ..., and finally, rule 'E4' is executed and the 'F' exit ('fail') is taken."

Next, 'SQUARE's marked "not accessible" are placed on the free list; see the three rules on page 'F'. Garbage collection is finally complete and execution of the program terminates. The program presented here would, in practice, be a subroutine of a larger program such as a LISP interpreter; it would be called when the free list was exhausted and it would return with the free list replenished. Thus the only output of this example is the state of the user's data after execution. We, therefore, include below a listing of the running of 'lispgc' followed by a use of 'agd', the AMBIT/G debugger. First, however, is included a diagram of the relevant user data after execution. Note that the running of this program takes approximately 500 seconds of CPU time.


SQUARE
TOP
P
SQUARE
3
SQUARE
6
SQUARE
1
D
A
SQUARE
5
SQUARE
BOT
B
SQUARE
4
SQUARE
2
B
C
B

```
→  hmu

   Multics 13.1, load 8.0/41.0; 7 users

   r 1312  .414  6+27

→  ambitg llspgc
   AMBIT/G


   r 1321  499.356  155+482



→  agd

→  DIAMOND P
   DIAMOND P:
             up/SQUARE TOP
             down/SQUARE TOP
→  SQUARE TOP
   SQUARE TOP:
             left/
             left-son/SQUARE 3
             down/SQUARE 1
             right-son/SQUARE 6
             right/
→  SQUARE 3
   SQUARE 3:
             left/
             left-son/SQUARE 1
             down/SQUARE 4
             right-son/CIRCLE D
             right/RIGHT-QUAD NA
```

(Cont' on next page)

```
→  SQUARE 1
   SQUARE 1:
                left/
                left-son/SQUARE BOT
                down/SQUARE 2
                right-son/CIRCLE B
                right/RIGHT-QUAD NA
→  SQUARE 6
   SQUARE 6:
                left/LEFT-QUAD RS
                left-son/CIRCLE A
                down/SQUARE BOT
                right-son/SQUARE 5
                right/RIGHT-QUAD A
→  SQUARE 5
   SQUARE 5:
                left/LEFT-QUAD RS
                left-son/SQUARE 4
                down/SQUARE 6
                right-son/SQUARE 2
                right/RIGHT-QUAD A
→  SQUARE 4;SQUARE 2
   SQUARE 4:
                left/LEFT-QUAD LS
                left-son/CIRCLE B
                down/SQUARE 5
                right-son/CIRCLE C
                right/RIGHT-QUAD A
   SQUARE 2:
                left/LEFT-QUAD RS
                left-son/SQUARE 4
                down/SQUARE 3
                right-son/CIRCLE B
                right/RIGHT-QUAD A
→  /q
   r 1324  4.770  60+106
```

ANOTHER GARBAGE COLLECTOR

Another garbage collector example is included here as the program 'mfgarb' (named for Michael Fischer). It uses the same basic technique as the previous example and therefore we will not present much discussion.

In this example, the initial graph is on the last page. The 'mark f' means "false" or "not accessible" and the 'mark t' has the opposite meaning. 'mark l' means "left", and 'mark r' means "right".

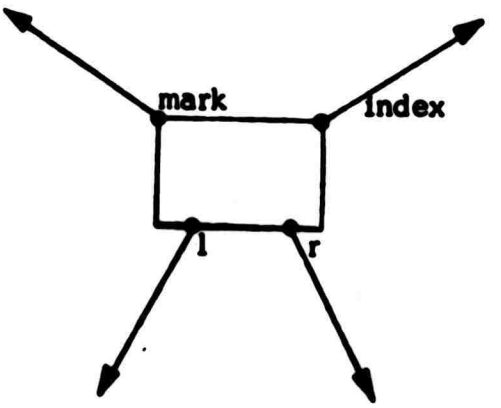
The result of running this program is a data graph where all 'square' nodes accessible from 'pointer p' have a 'mark t'.

Note that three of the pages do not have labels in the upper right corner. They are highly recommended, but are optional.

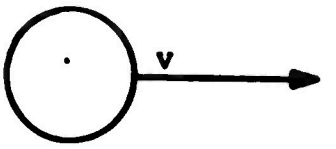
The running of this program takes less than four minutes of CPU time, as indicated in the following output from the terminal.

```
-> hmu
   Multics 13.0A, load 26.0/41.0; 25 users
   r 1440  .390  6+21
-> ambitg mfgarb
   AMBIT/G

   r 1446  230.020  324+461
```

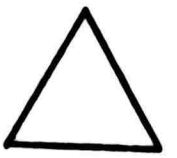
mfgarb



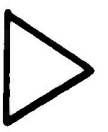
pointer :



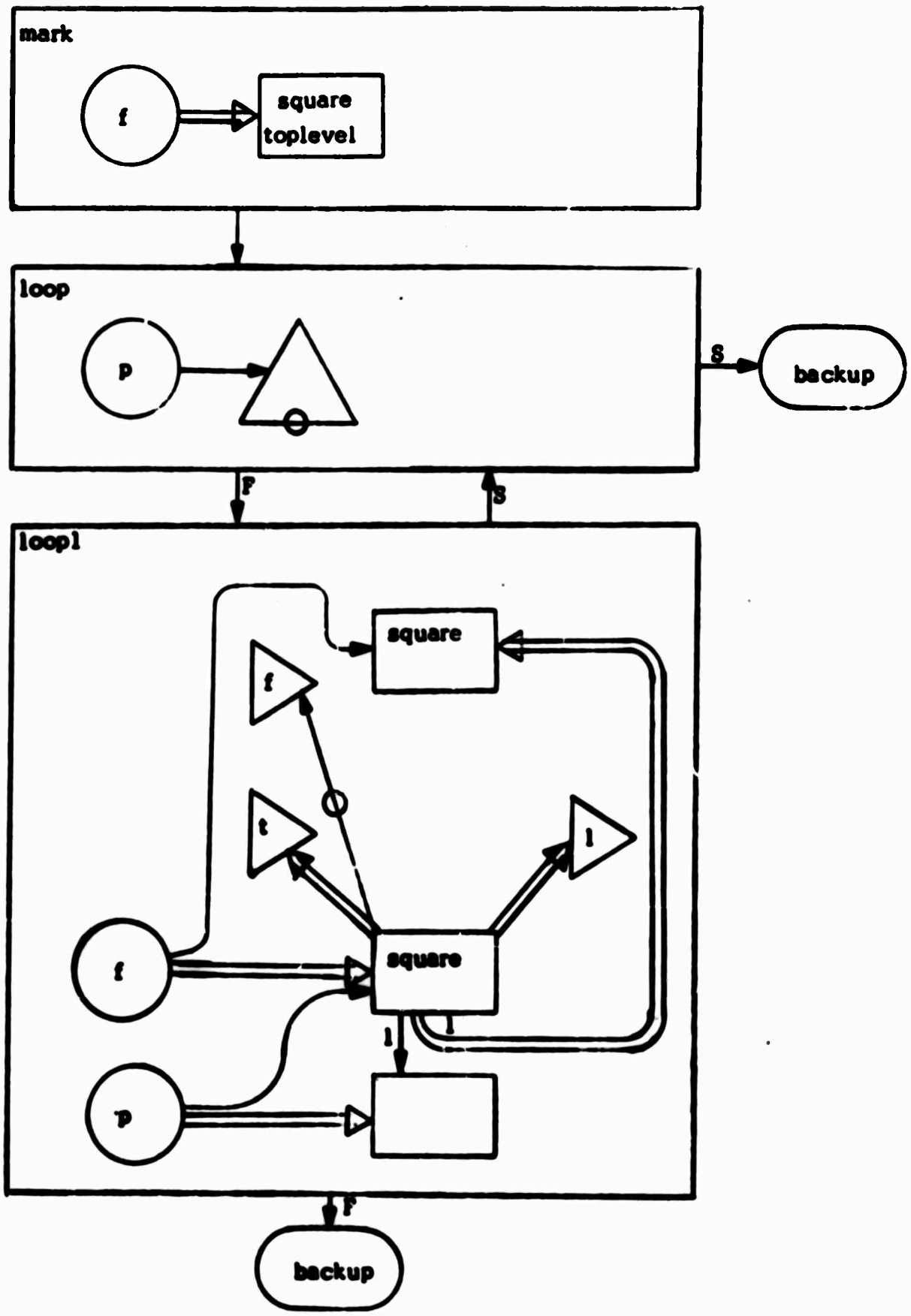
atom :

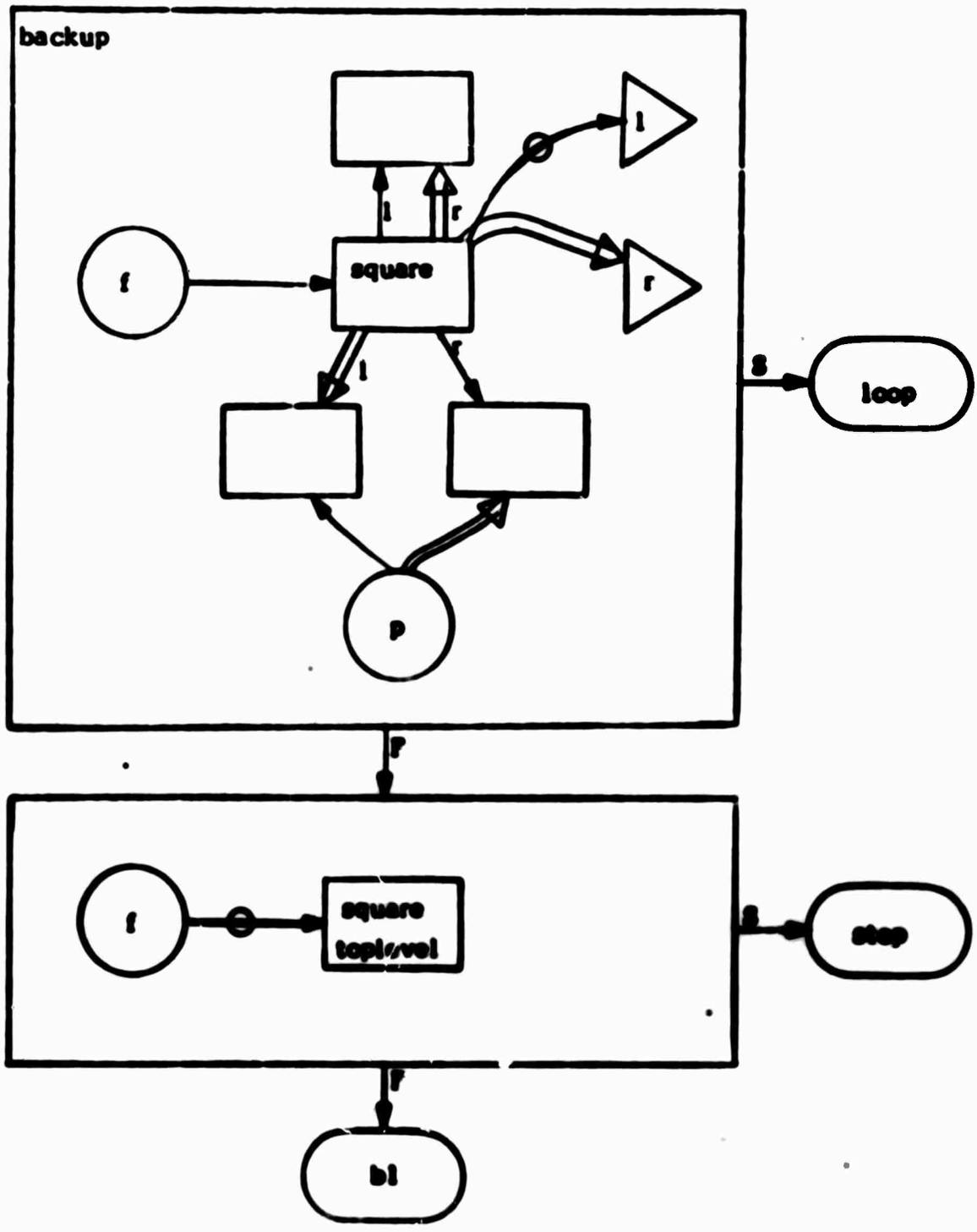


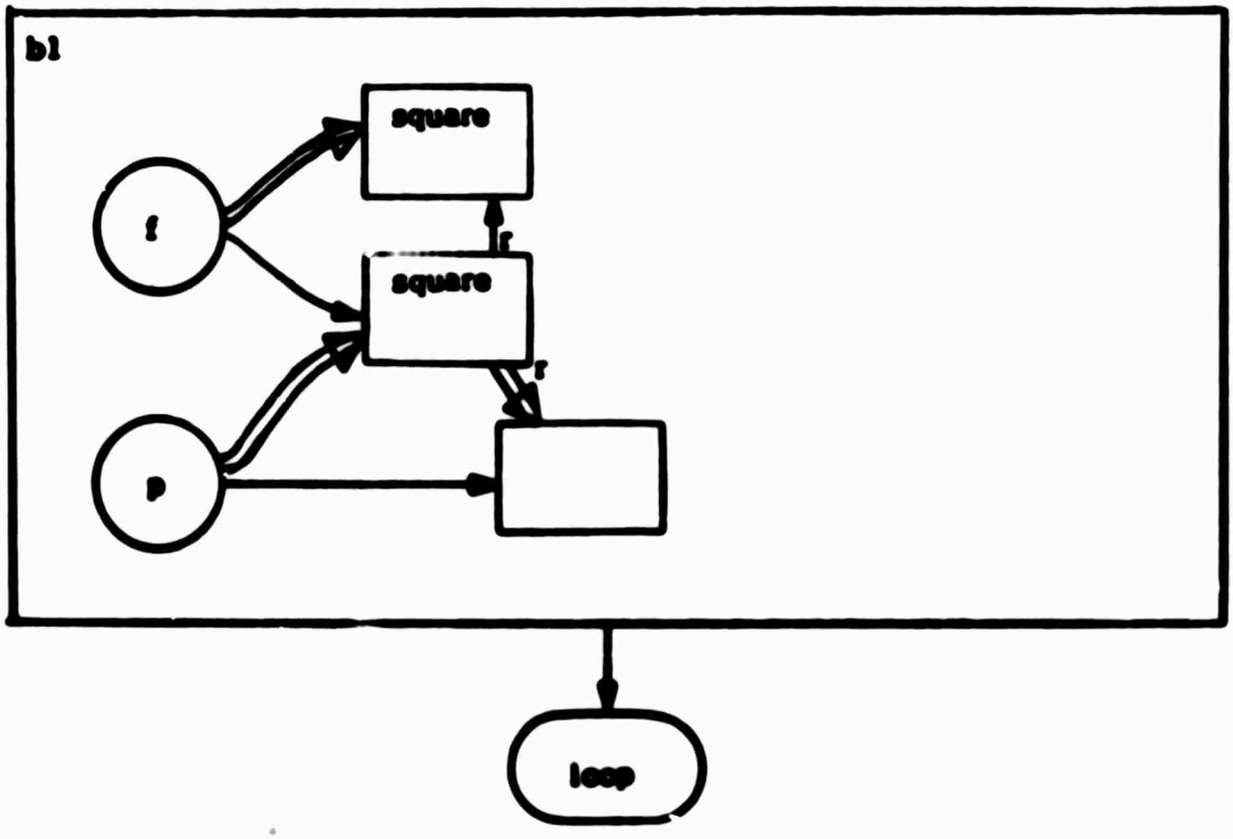
mark :

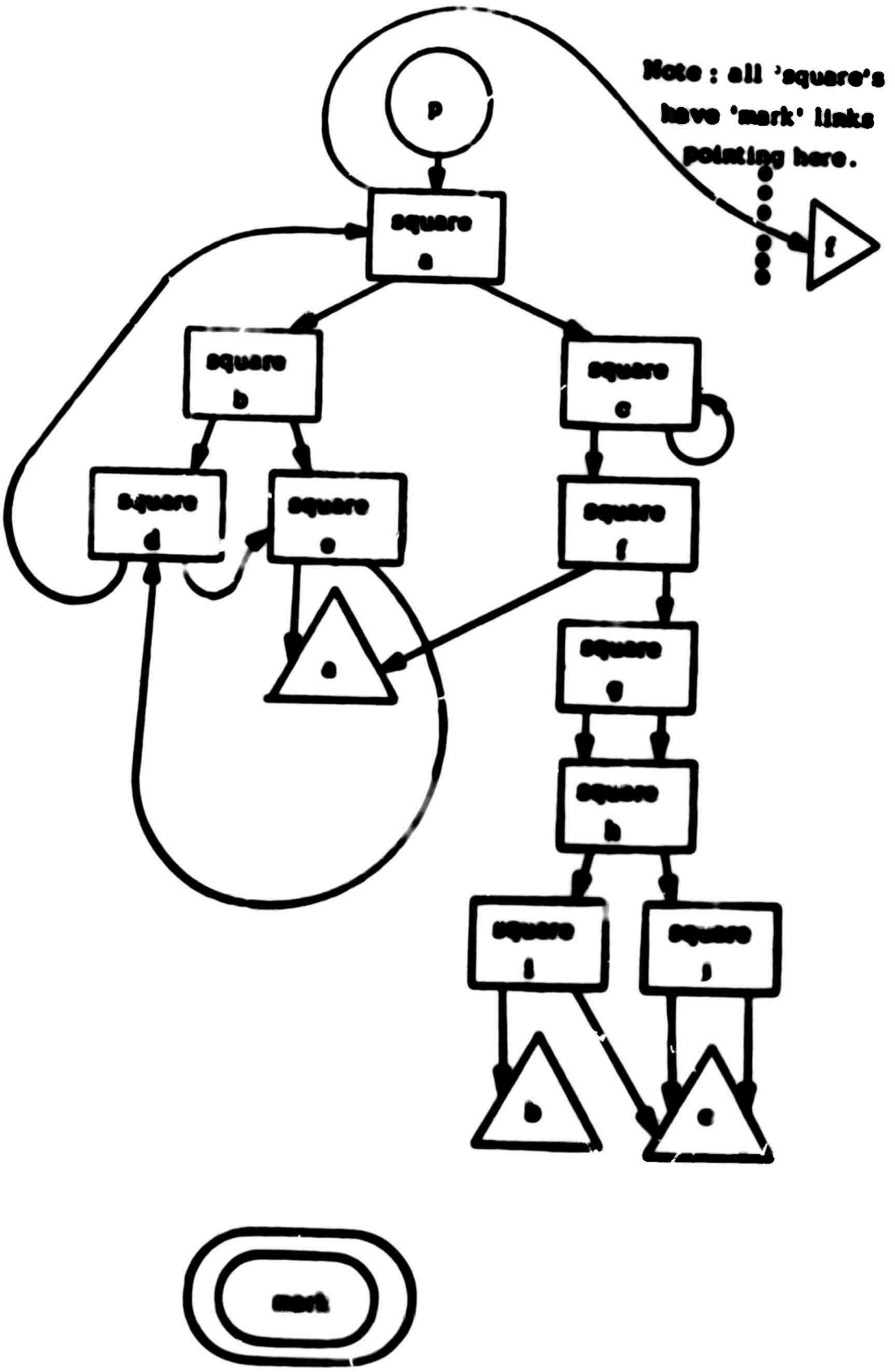


start of program




backup
l
r
square
f
l
r
s
loop
p
F
f
square
topl/vel
s
stop
F
bl


b1
square
f
square
r
r
p
loop


Note : all 'square's
have 'mark' links
pointing here.
p
square
a
square
b
square
c
square
d
square
e
square
f
square
g
square
h
square
i
square
j
mark

AN INTERACTIVE PROGRAM

This AMBIT/G program, 'octdec', accepts an octal number typed by a user on the terminal and types back its decimal equivalent to the user. The program begins by prompting the user with a ' > ' on a new line. The user is then expected to input some octal number and terminate with a new line (carriage return). If he types a '$', execution will terminate. The receipt of any unexpected character causes the program to ignore the typed line and restart by prompting the user.

If a sequence of octal digits is typed, the program converts the digits to the integer which is named by those digits (by a call on the built-in 'locate'). The program removes leading zeros during this phase and also prints back the typed number with leading zeros removed (indented by a tab). Note that this integer is a strange value*, but it can be converted arithmetically to another integer which has the value of the octal number which the user typed. This is done entirely, except for initialization, in the large rule on page 'oct.6'. The one rule demonstrates the superiority of the AMBIT/G diagram in expressing certain algorithms. Here is a case where three variables are updated on each execution of the rule, and their old and new values are interrelated in a complicated way. We urge the reader to study this glorious rule.

Finally, the program prints the decimal number (after a tab) and then restarts to prompt the user.

Although 'octdec' is an interactive program, it takes around half a minute to respond with an answer after the user inputs his number. The following terminal listing of a run of 'octdec' demonstrates its use and shows how much time it takes. Note the first complete run took nearly ten minutes of CPU time, which corresponds to a cost of about $70 during the prime shift. The listing includes two instances where 'rgo' was used to restart the execution of the program at 'rule prompt-user'. Finally, 'octdec' was given too large a number, and it terminated with an error message. The last file for this run is also listed to explain the cause of the error.

---

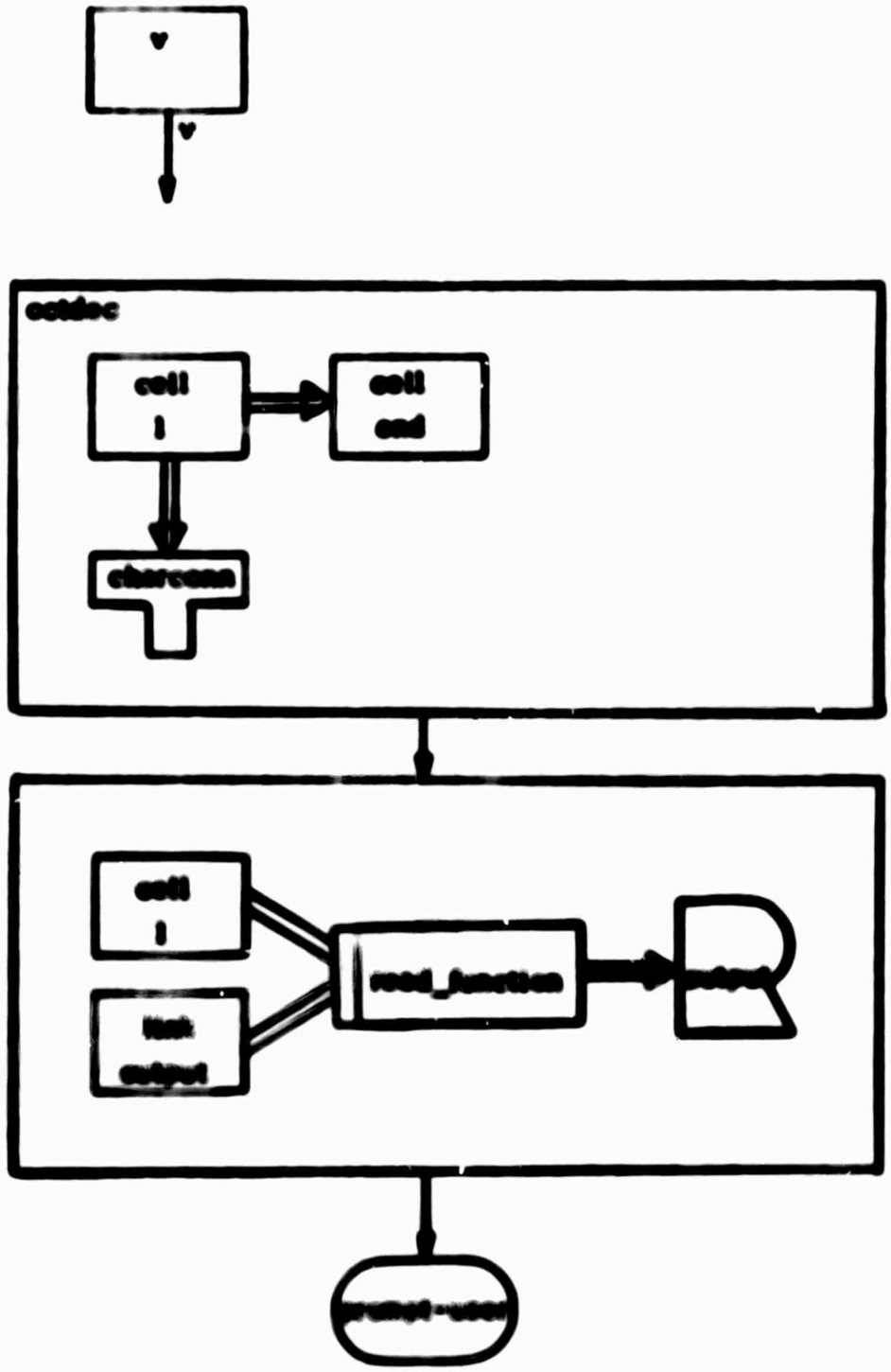
* This "strange" value is the number whose decimal representation has the same digits as the octal number typed in.

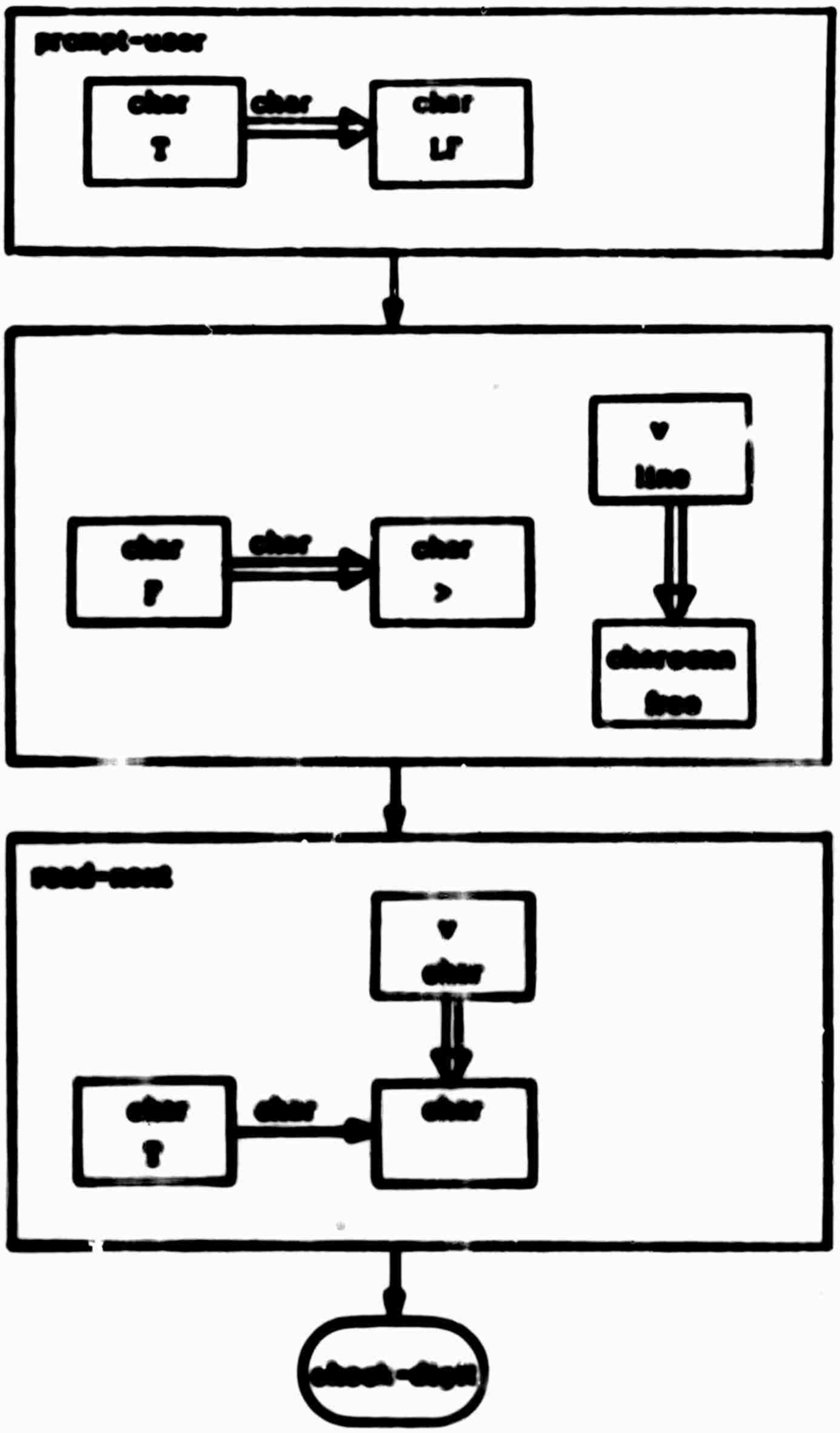
```
-> hmu

Multics 13.1a, load 11.0/41.0; 10 users

r 2547  .601  4.19

-> ambitg octdoc
AMBIT/G


-> >123
                123         83

-> >10
                10          8

-> >00500
                500         320

-> >18

-> >-5

-> >0
                0           0

-> >8
r 11  591.504  121.850

-> eqd

-> /w circle r/volum/prompt-user
circle r1
                volum/rule stop
BECOMES         volum/rule prompt-user
-> /R
EXECUTION CONTINUES AT "rule prompt-user"

>
-> >654
                654         428

-> >8
r 13  69.276  36.165
```

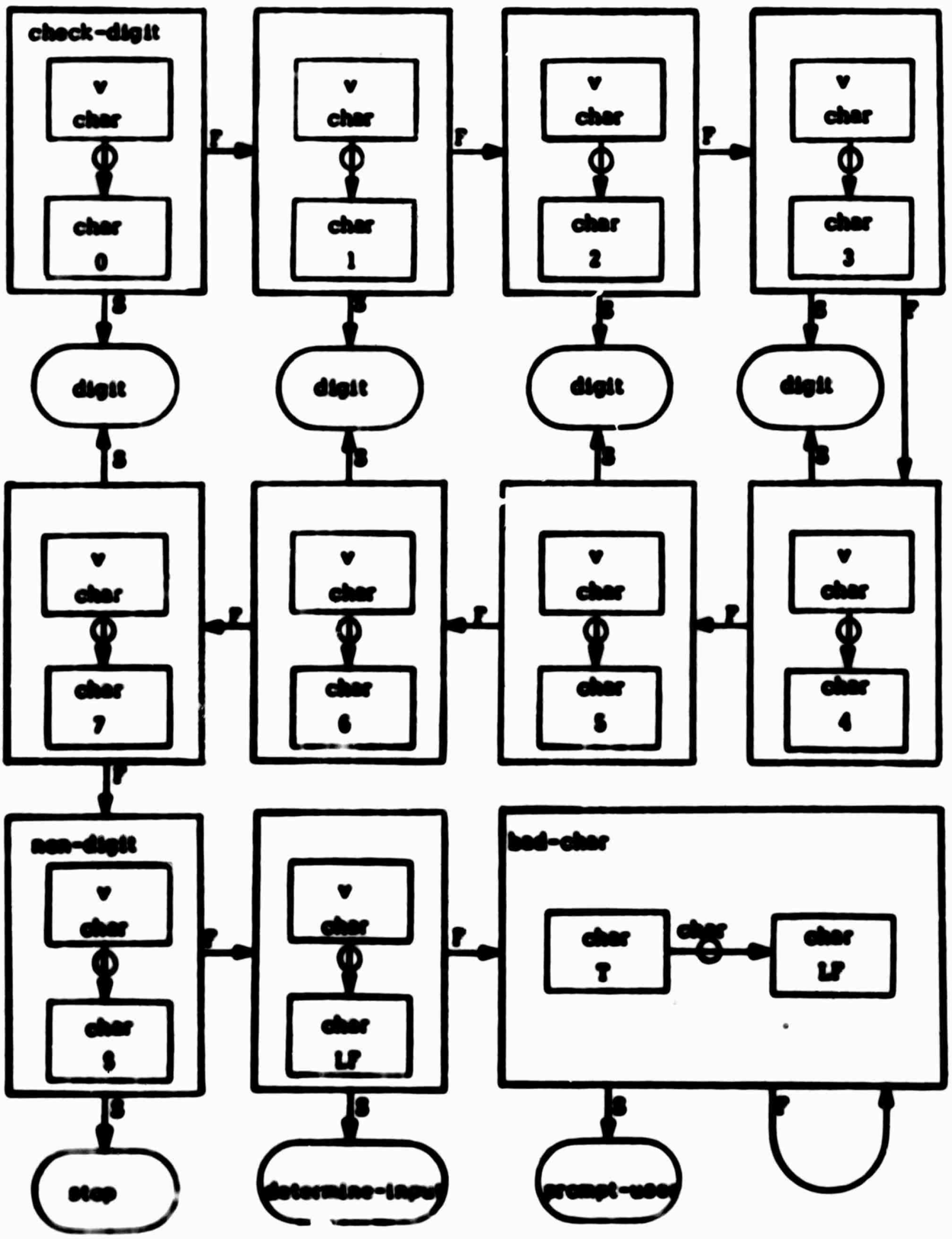
(Cont' on next page)

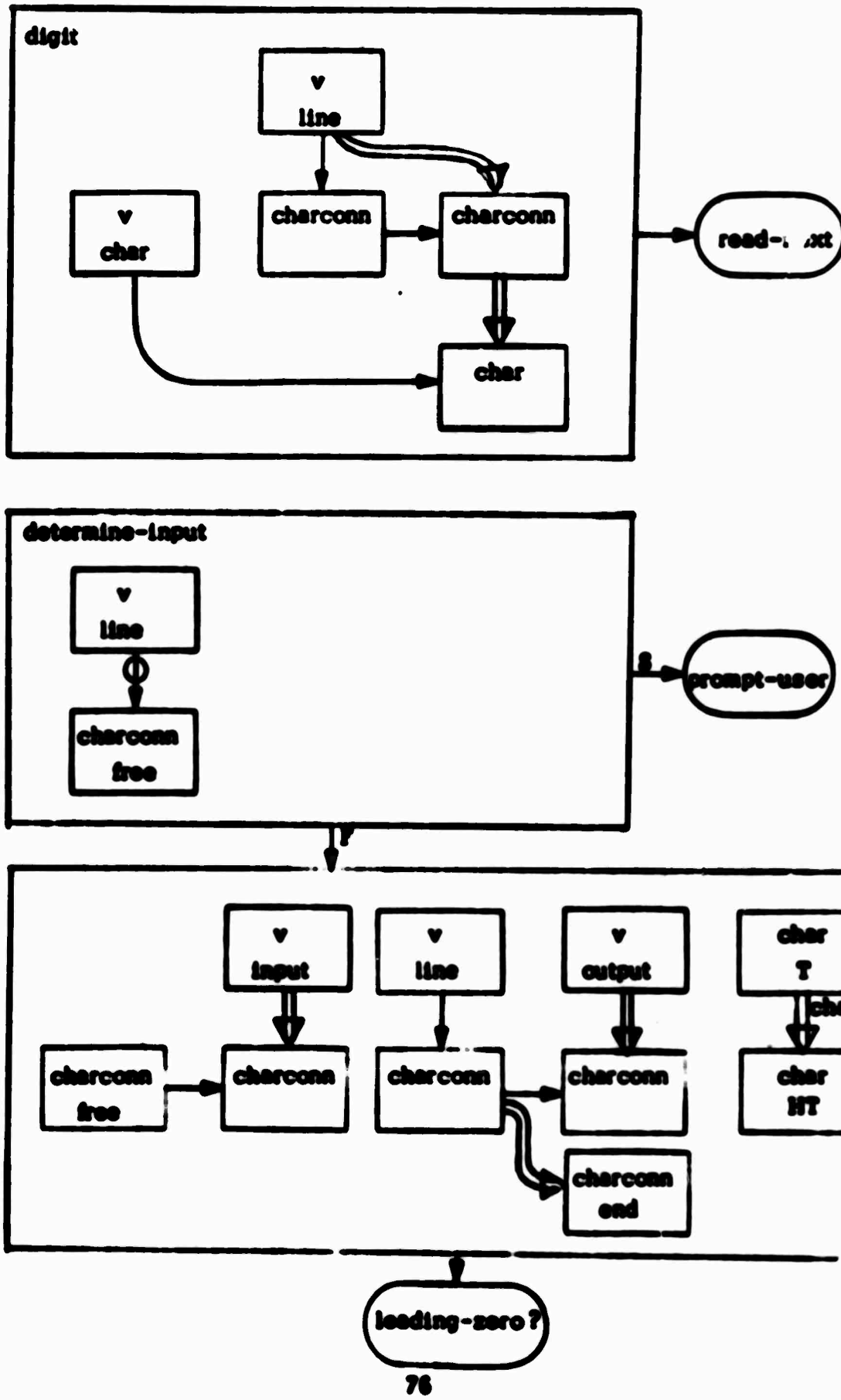
```
/fagd

/w circle r/value/prompt-user
circle r1
              value/rule stop
BECOMES       value/rule prompt-user
/G
EXECUTION CONTINUES AT "rule prompt-user"

>
>9999

>10000

AMDIT/6 Error:       A call on the builtin "multiply" produced a product
        of "32768", which is outside of the range of integers.
This error occurred while interpreting the rule "rule convert".
The interpreter was processing the link  "(integer 4096, integer 8)   multiply   (   )".

r 16  45.830  51+195

pr (cstdnc.hints cstdnc.ambig)

                cstdnc.hints        12/31/70  0017.8 est Thu

largest_integer=10000

            30        1
```

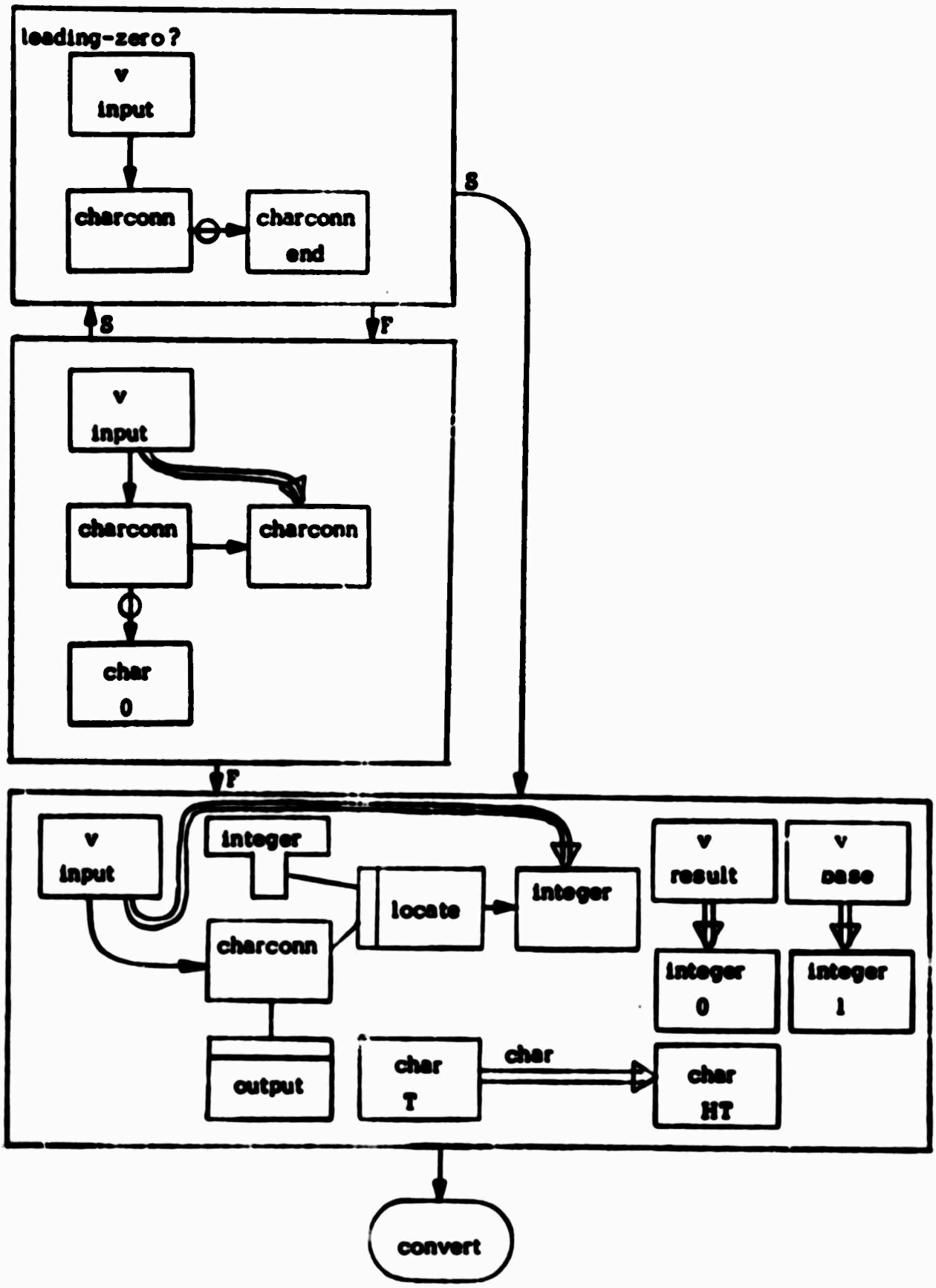
octdec

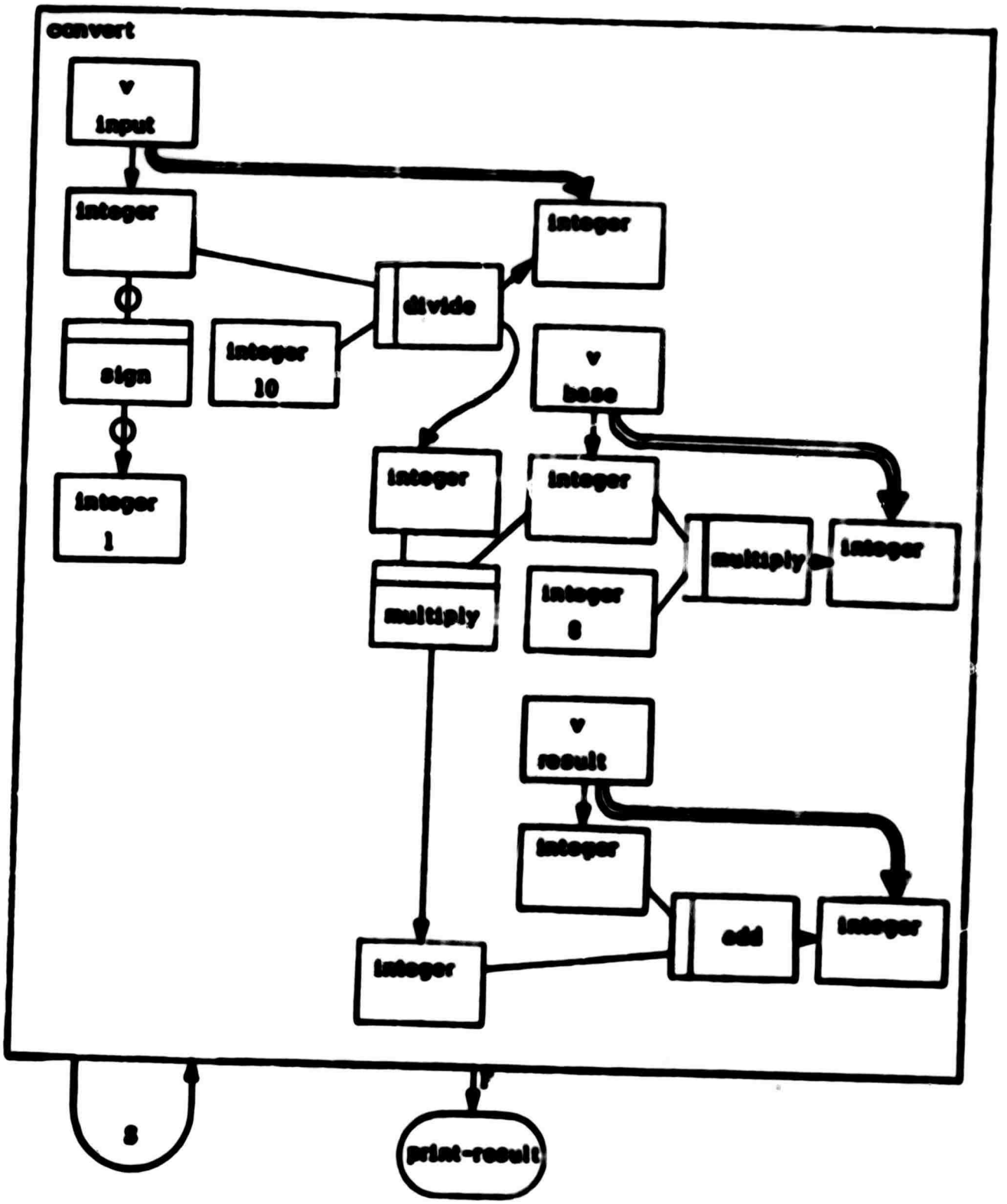


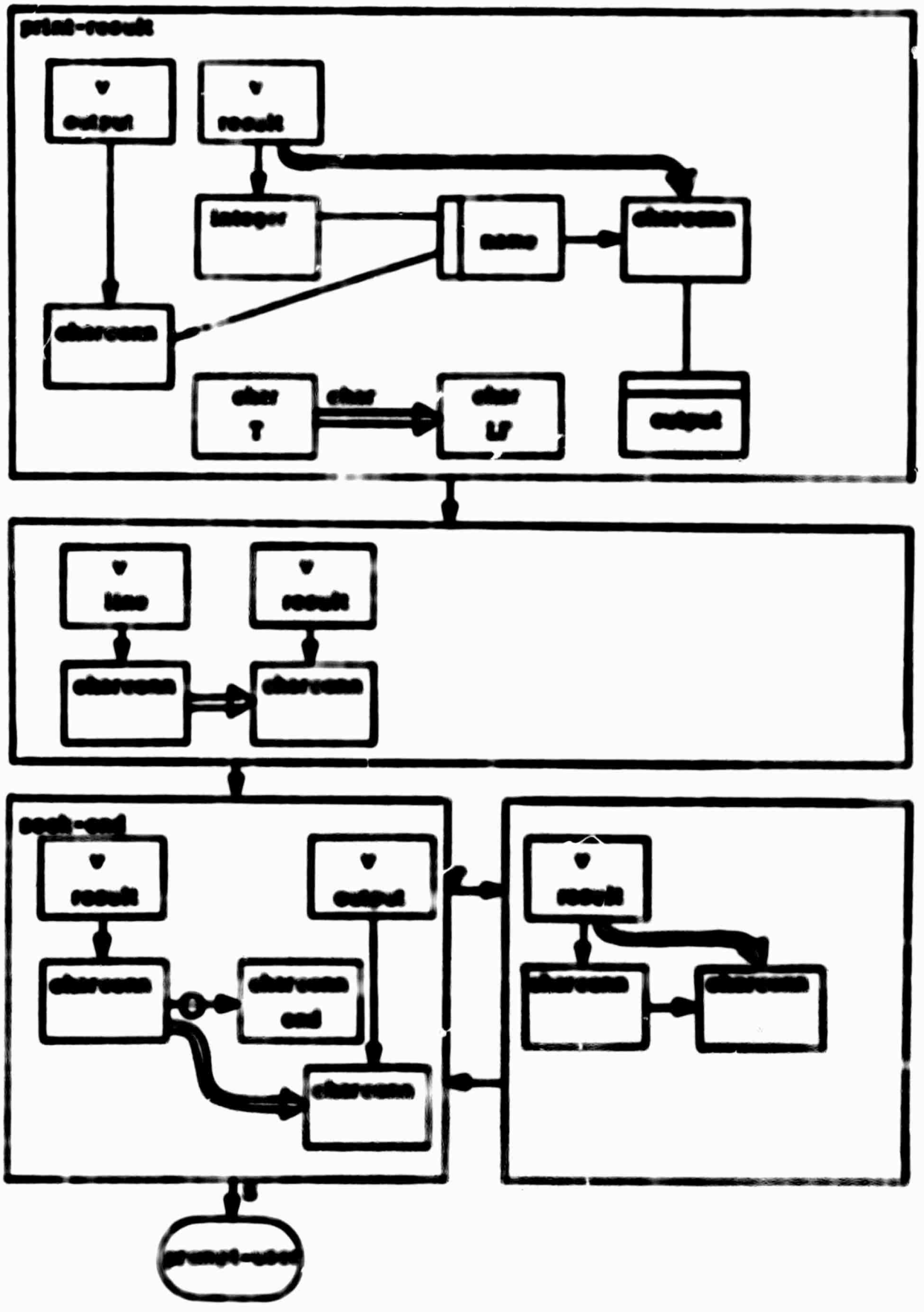

prompt-user
char
char
char
LF
char
P
char
char
>
line
charconn
free
char
char
char
char
check-digit

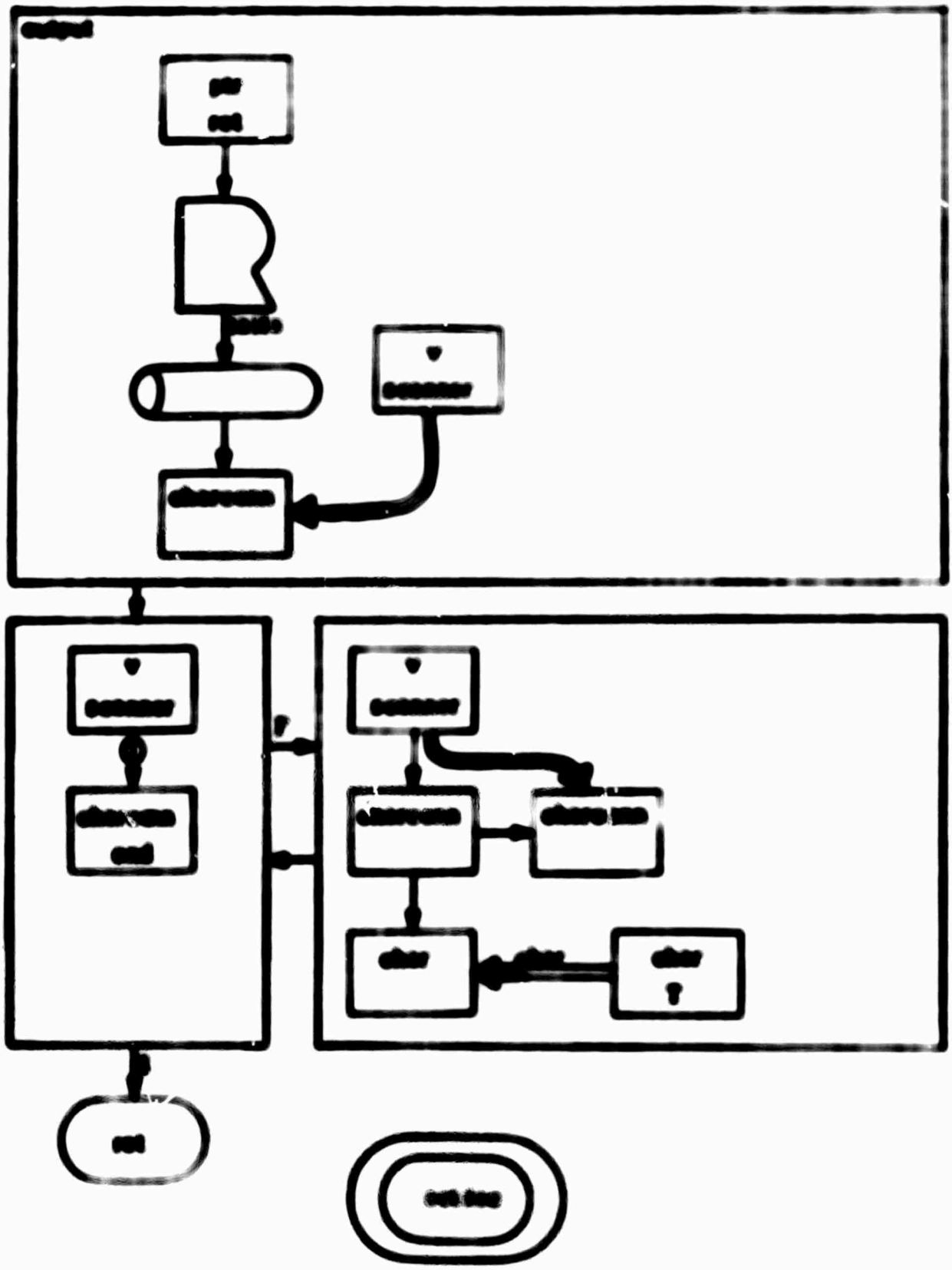

check-digit
char
char 0
char 1
char 2
char 3
digit
char 7
char 6
char 5
char 4
non-digit
char $
bad-char
char ?
char LF
stop
prompt-user


digit
v
line
v
char
charconn
charconn
char
read-
xt
determine-input
v
line
charconn
free
prompt-user
v
input
v
line
v
output
char
T
char
charconn
free
charconn
charconn
charconn
char
HT
charconn
end
leading-zero?


leading-zero?
v
input
charconn
charconn
end
S
S
F
v
input
charconn
charconn
char
0
F
v
input
integer
locate
integer
charconn
v
result
v
base
integer
0
integer
1
output
char
T
char
char
HT
convert


convert
input
integer
integer
divide
sign
integer
10
base
integer
integer
integer
1
multiply
integer
integer
8
result
integer
add
integer
integer
8
print-result


print-result
output
result
integer
name
character
character
char
'
char
LF
output
line
result
character
character
result-out
result
output
character
character
character
result
character
character



[redacted]

The "quicksort" program* operates on a doubly-linked list of integers connected by nodes of type 'c'. The node 'c 0' is used to terminate each end of the list. The connections form an "engraved" list so that only the destination of the 'v' link of each 'c' may change. Thus the connections create a kind of hardware memory where there is a predecessor and successor function. Also engraved is a doubly-linked stack of 'stack' nodes, where each stack element can save two 'c' nodes. Page 'ps.3' of the program provides the established stack and lists manners, the last contains an initial out-of-order sequence of integers.

The purpose of the program is to sort the list of integers into a monotonically increasing sequence by exchanging pairs of integers. It operates recursively and uses the stack to save subsets. The general scheme is to split the input list into two subsets. The left or low one includes integers less than some arbitrarily chosen integer, and the right or high subset includes integers greater than that chosen integer. Once this is done, that same scheme can be applied to each subset, and so on.

In the "quicksort" program, at the beginning of any stage in the sort, 'p l' points to the low end, and 'p h' points to the high end of the given subset. 'p lf' is used to walk from 'p l' up towards 'p h', and 'p hf' is used to walk from 'p h' down towards 'p l'. The chosen integer is the first element of the subset (i.e., selected by 'p l'); it is pointed to by 'p boss'. We first come up the list using 'p lf' to attempt to establish a region between 'p l' and 'p lf' where all integers are less than or equal to the chosen one. If this pointer gets to an integer outside of the region, the up-scan ceases. Then a similar down-scan is done from 'p h' using 'p hf'. If the high temporary pointer reaches the low one, the splitting into subsets is complete. If, otherwise, 'p hf' arrives at an integer outside of the upper region, the down-scan ceases

---

* The algorithm used in this program is an adaptation of the steps in a paper by Thomas N. Hibbard in the [redacted], vol.9, no.3 (May 1963) pp.266-312, [redacted] "An Empirical Study of Minimal Storage Sorting".

and the two temporarily selected integers are then swapped. Then both regions are given for further extension. A careful inspection of the program will reveal that elements equal to the chosen integer can end up in either region.

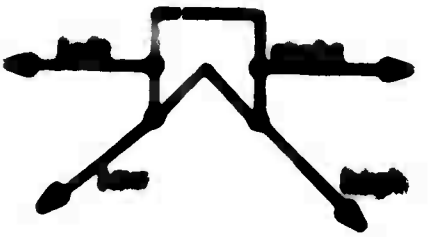
Once the given subset is properly split, the ends of the upper subset are pushed onto the stack as a deferred task, and the lower subset is sorted. A deferred task is popped off the stack when a subset to be sorted is composed of only one element. The interested reader should now be able to deduce the details omitted in the above description by studying the "quicksort" program.

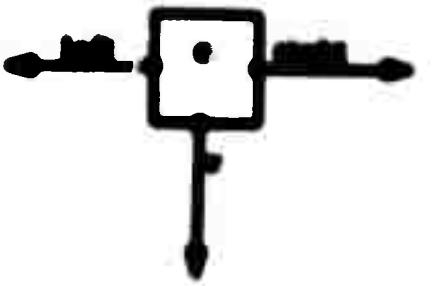
Given below is a terminal listing of a run of "quicksort". Note that it takes just under 100 seconds of CPU time. Following the run is a use of "sgl" to confirm that the list is sorted properly.

```
—> how
    Multics 23.1a, load 30.0/63.0; 9 users
    r 2225  .211  0+7
—> [illegible] quicksort
    [illegible]

    r 2226  [illegible]  37+1013
—> sgl
—> c a/v
   c a1
              v/integer 1
—> c b/v
   c b1
              v/integer 2
—> c [illegible]/v
   c c1
              v/integer 3
—> c d/v
   c d1
              v/integer 6
—> c e/v
   c e1
              v/integer 7
—> c f/v
   c f1
              v/integer 8
—> /q
   r 2227  3.239  [illegible]
```

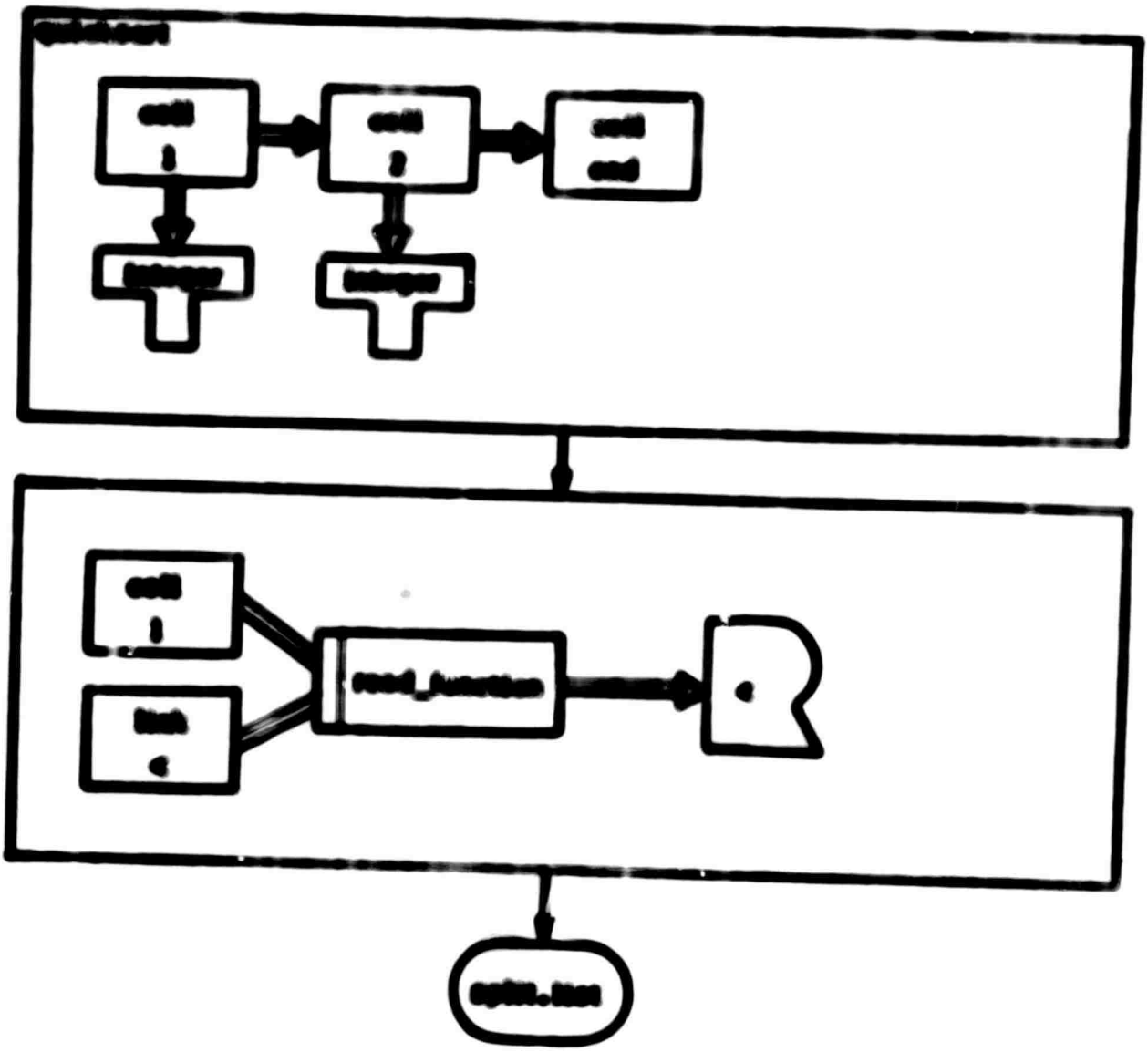
[redacted]

[redacted]

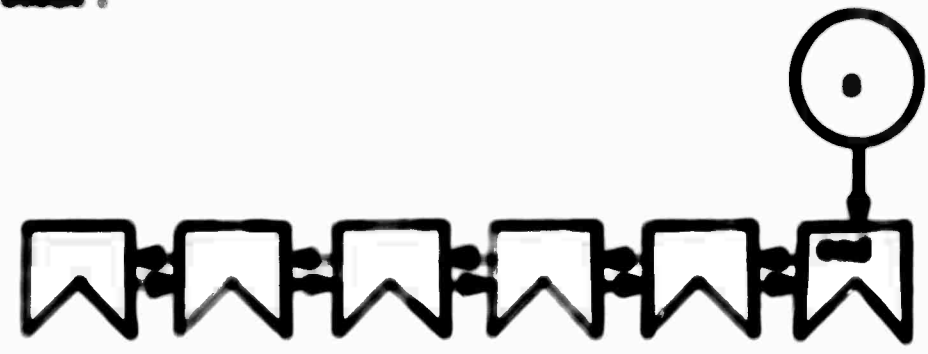




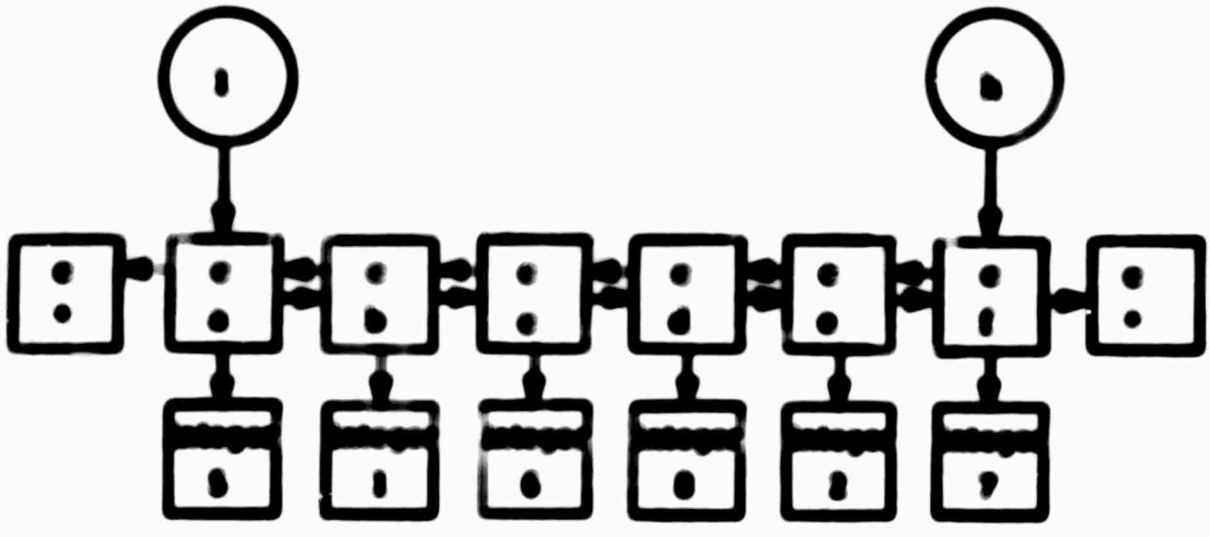
[redacted]

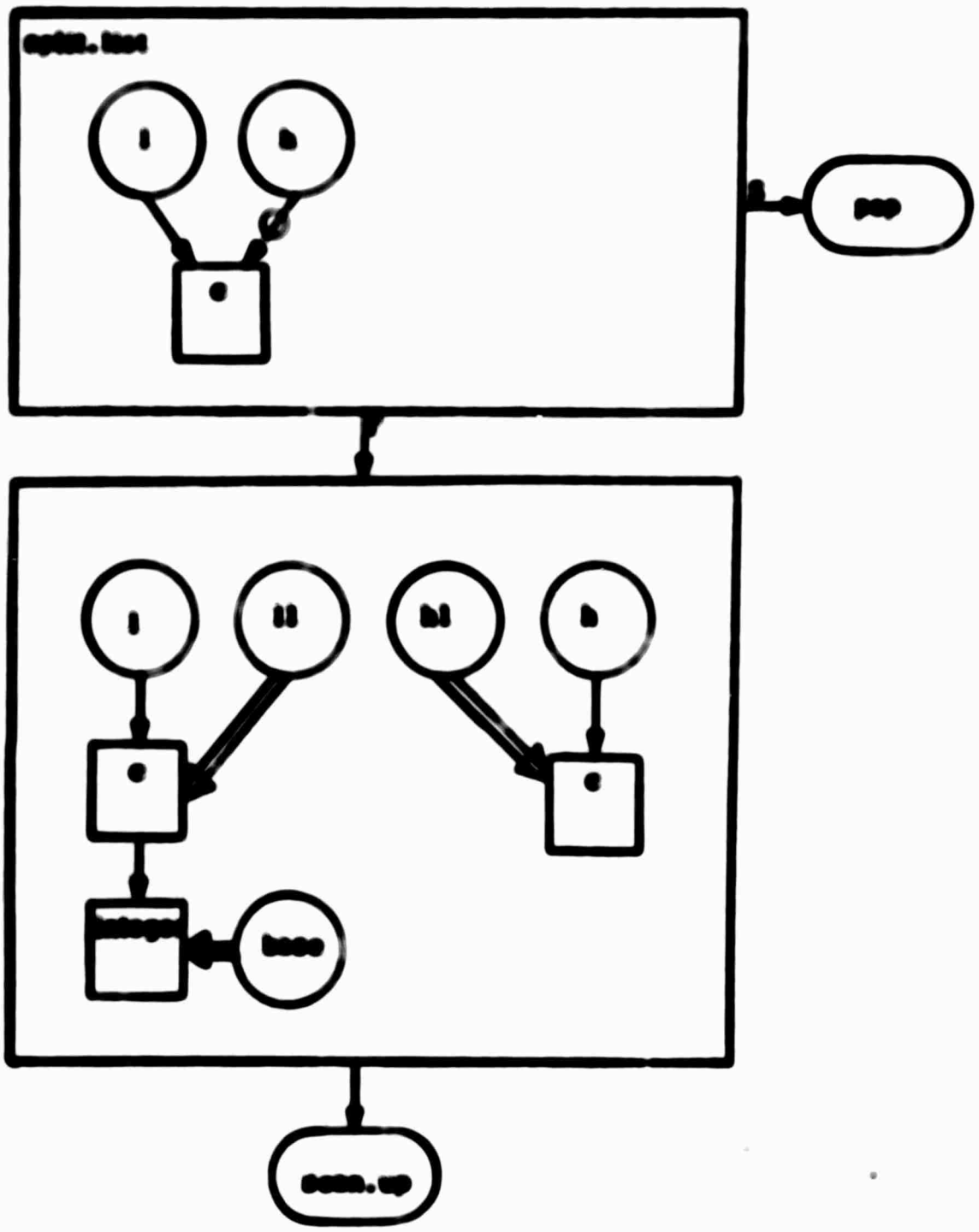


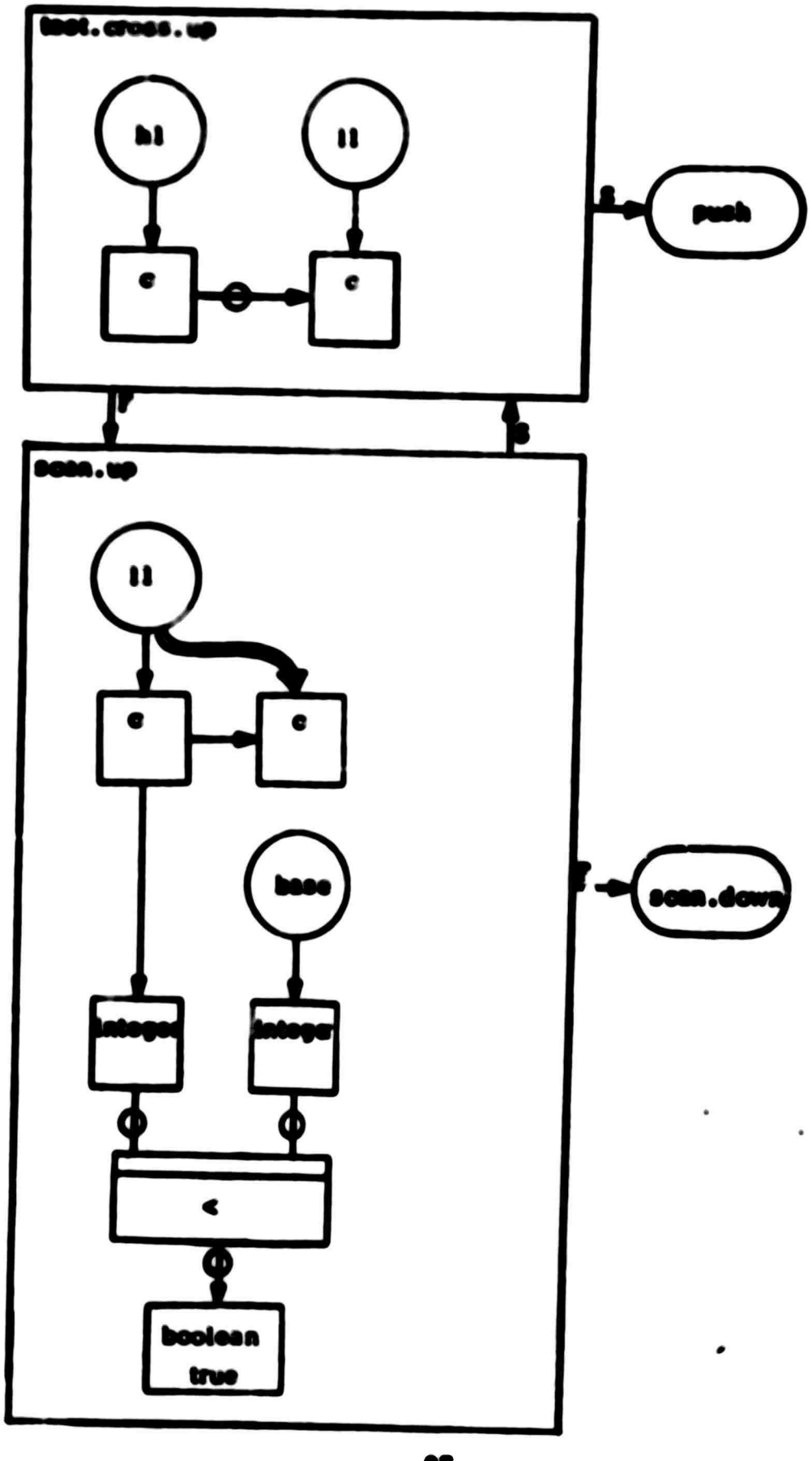
stack :

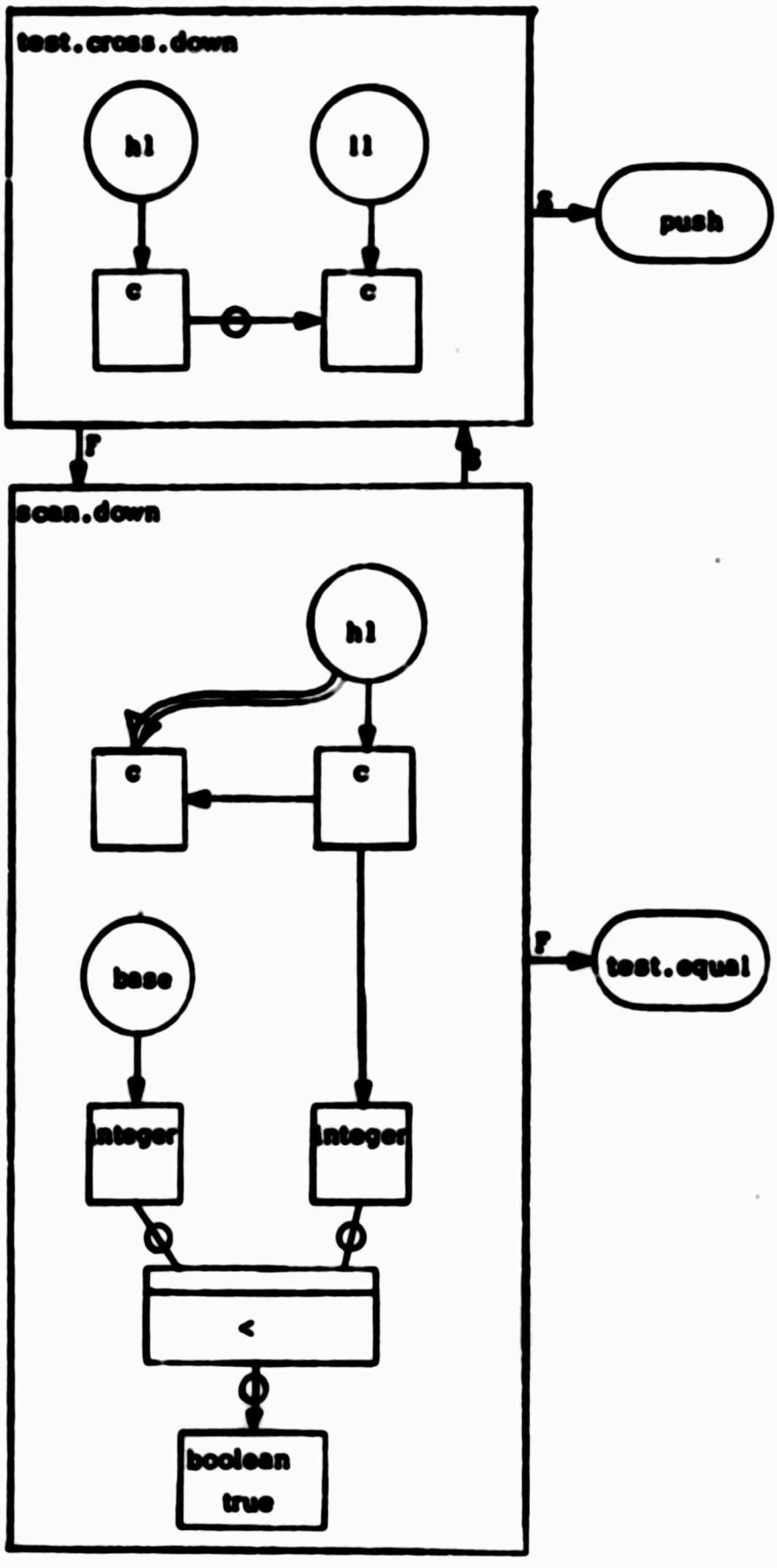


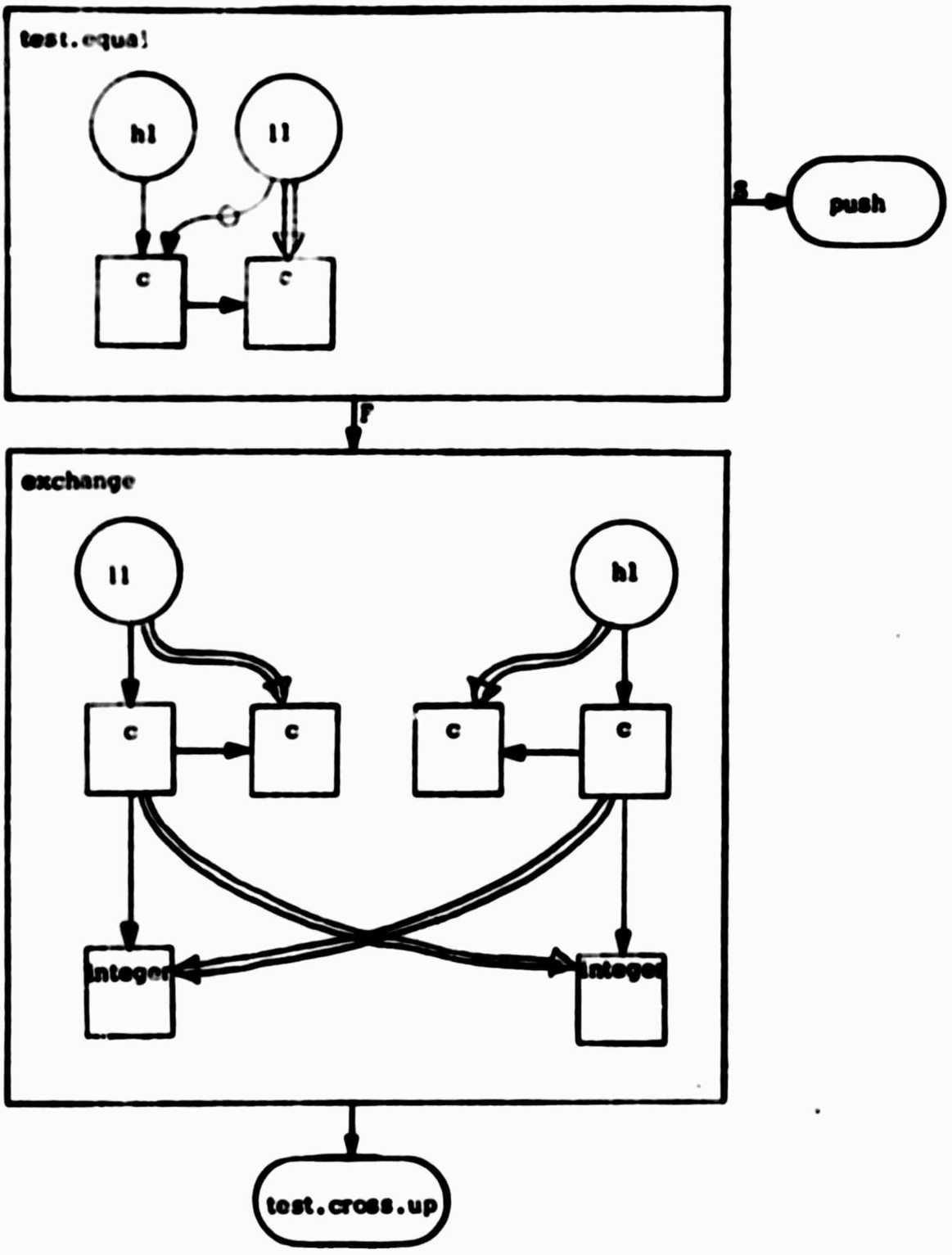
input bus :

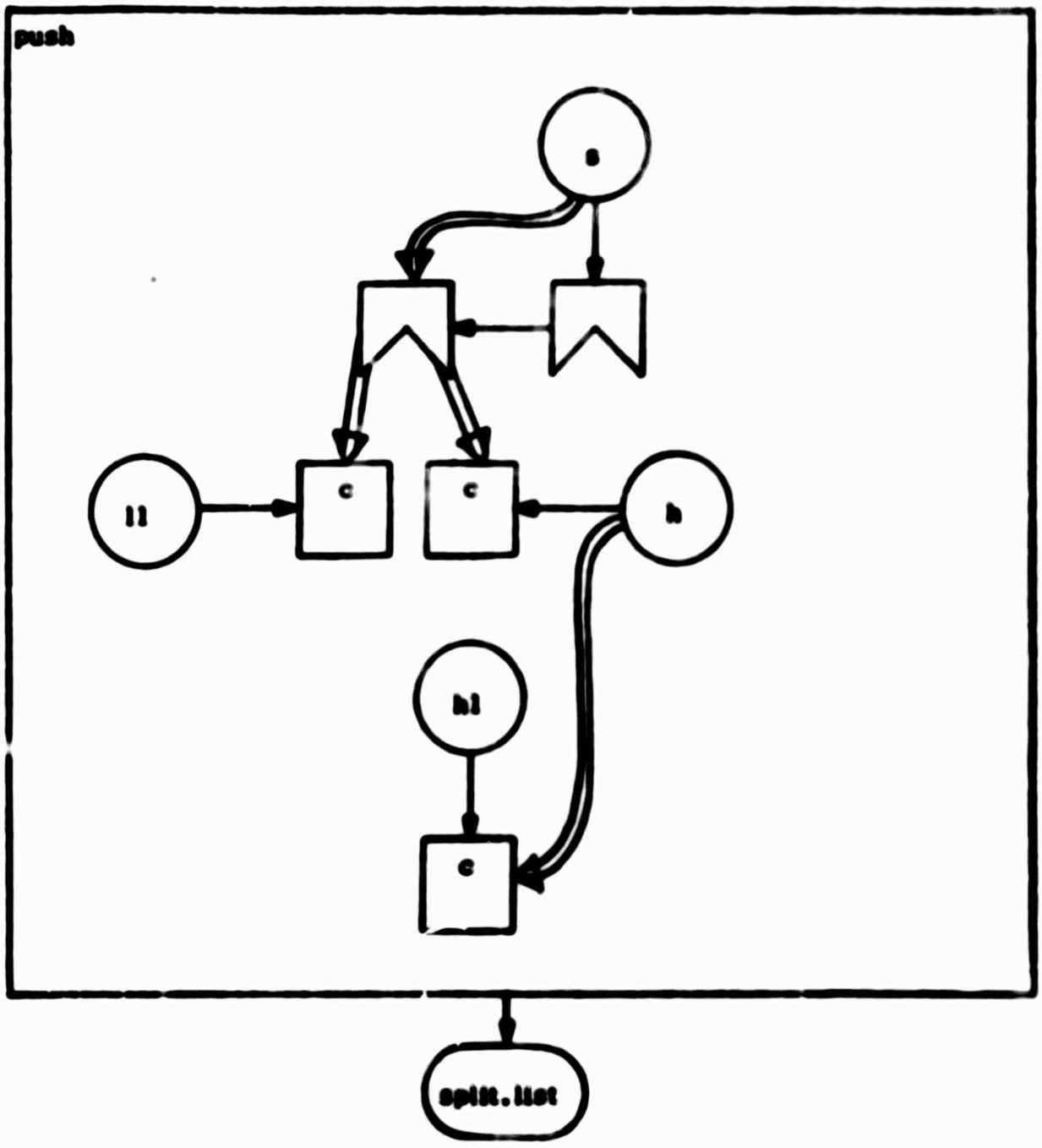


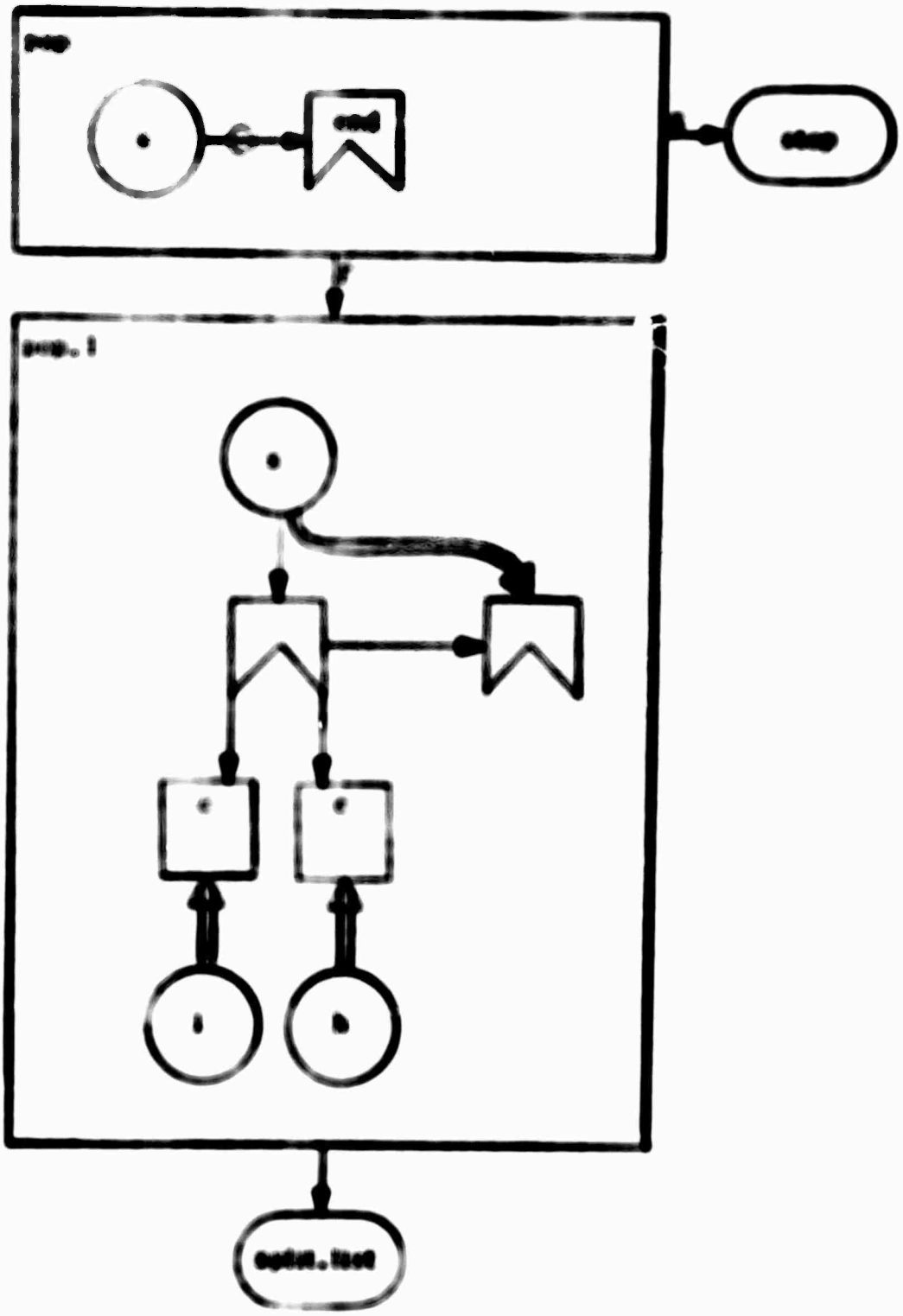


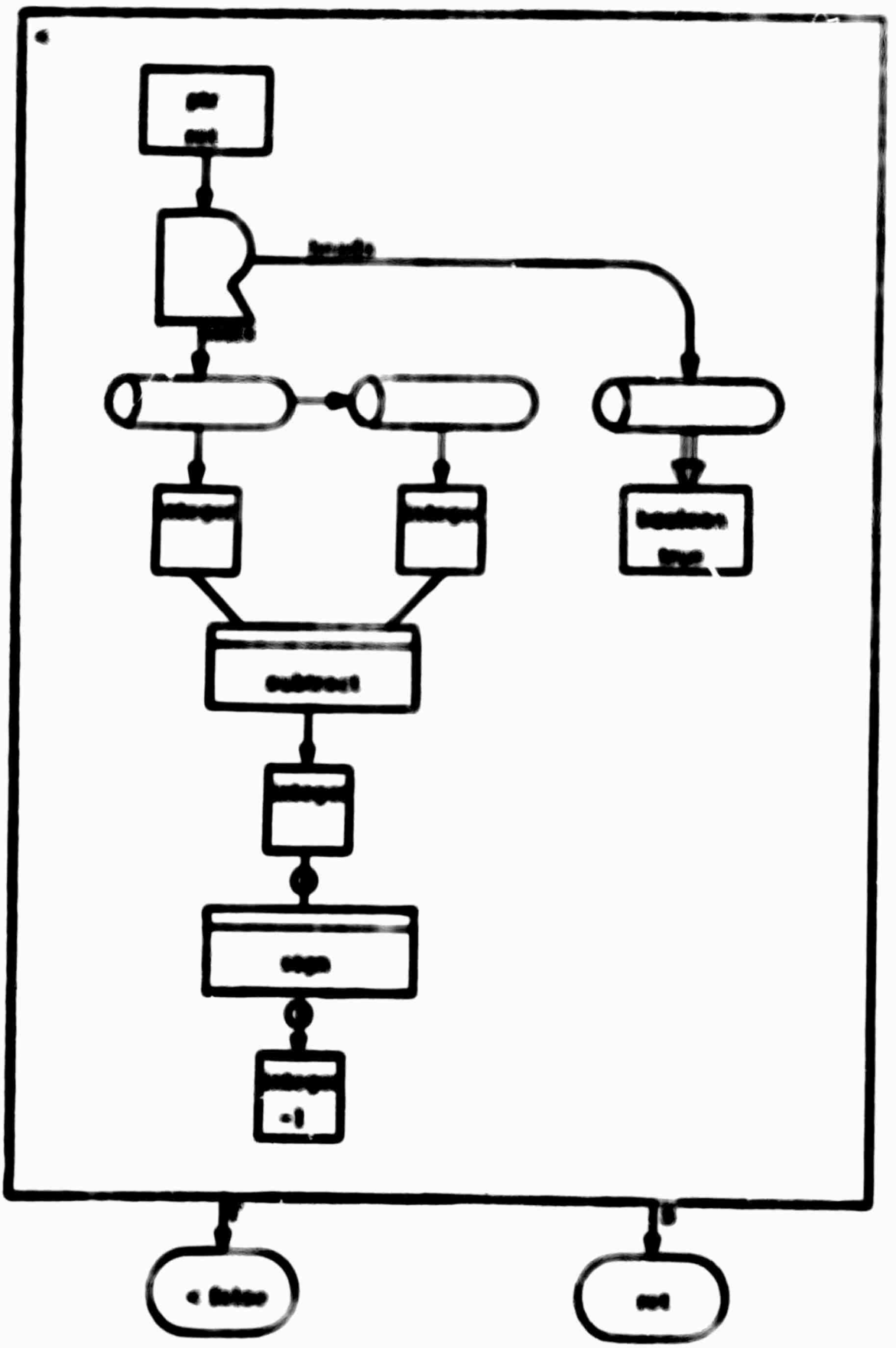

boot.cross.up
h1
l1
c
c
push
f
s
scan.up
l1
c
c
base
integer
integer
<
boolean
true
scan.down

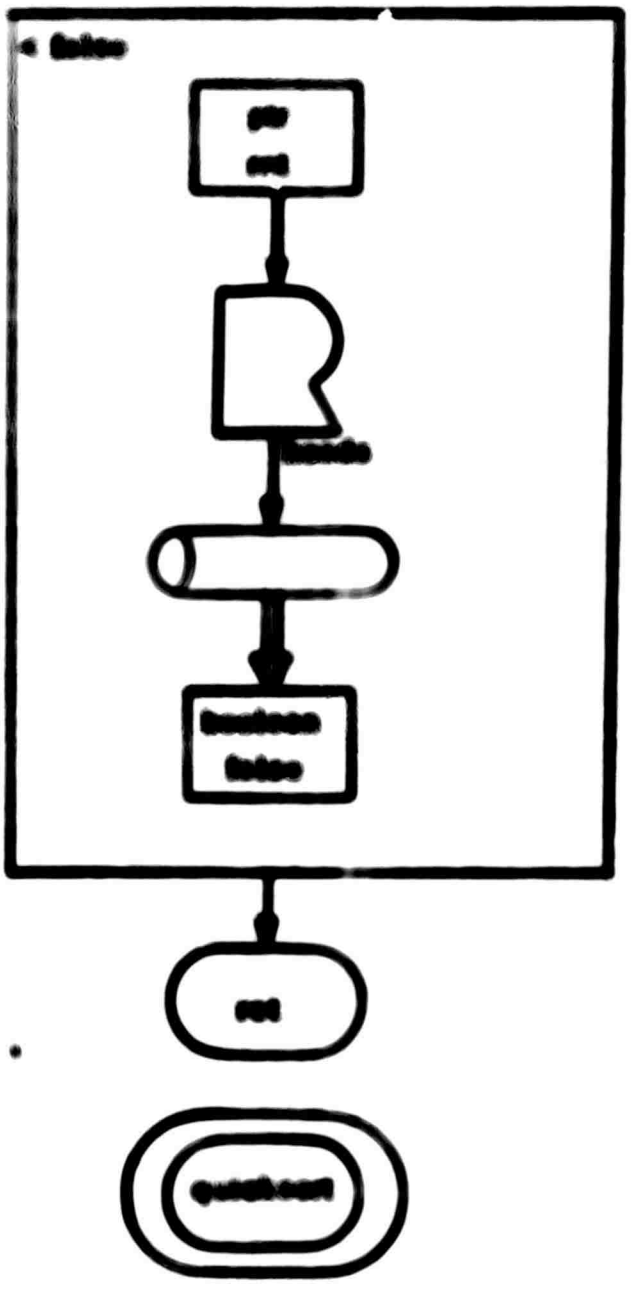

test.cross.down
hl
ll
c
c
S
push
F
S
scan.down
hl
c
c
base
integer
integer
<
boolean
true
F
test.equal


test.equal
hl
ll
c
c
s
push
f
exchange
ll
hl
c
c
c
c
integer
integer
test.cross.up


push
a
c
c
h
hl
c
split.list







FACTORIAL COMPUTATION AND RECURSION

This example is included to demonstrate both a recursive function definition of a factorial computation and the package of functions for supporting recursion. The whole program, 'fact', computes the factorial of the integer 4 that will be on page '1.1' -- the eighth page of the listing. The definition of the factorial function is given on the very next page of the listing. Note the use of '[illegible]' to fetch the argument and '[illegible]' to return the result. These are argument and result passing functions for use in recursively called functions.

Although the AMPL/G interpreter does not include the capability for performing recursive function calls, it does have "handles" for augmenting the interpreter to accommodate recursion: the built-in nodes 'pir rec' and 'pir [illegible]' and the replaceable built-in rules 'go', 'ret', and 'help'. Most of the 'fact' program consists of a general package of functions for supporting recursion which do replace these three built-in rules with somewhat complicated routines. We recommend that the reader must have a clear understanding of the function calling mechanism of the interpreter in order to benefit by studying this example, other than the 'factorial' function itself.

The recursion package defines three additional types of nodes which it uses to save otherwise lost information on a stack headed by 'pir call_stack'. The elements of this stack are of type '[illegible]'. The '[illegible]' link of '[illegible]'s is used to save essential information about re-entered rules in the user program by pointing to a list of '[illegible]' nodes. The '[illegible]' link of '[illegible]'s is used to point to a list of '[illegible]' nodes which are used to save the [illegible] of a re-entered rule.

The node 'pir rec' is a switch to indicate whether recursive handling is ON ('flag yes') or OFF ('flag no'). The switch is needed since the recursion package itself uses the function calling mechanisms of the interpreter; without it or an equivalent switch the package would recursively loop.

When the AMPL/G interpreter processes a user function call, it sets 'pir ref' to the 'rule' in which the call is made and the '[illegible]' link of that

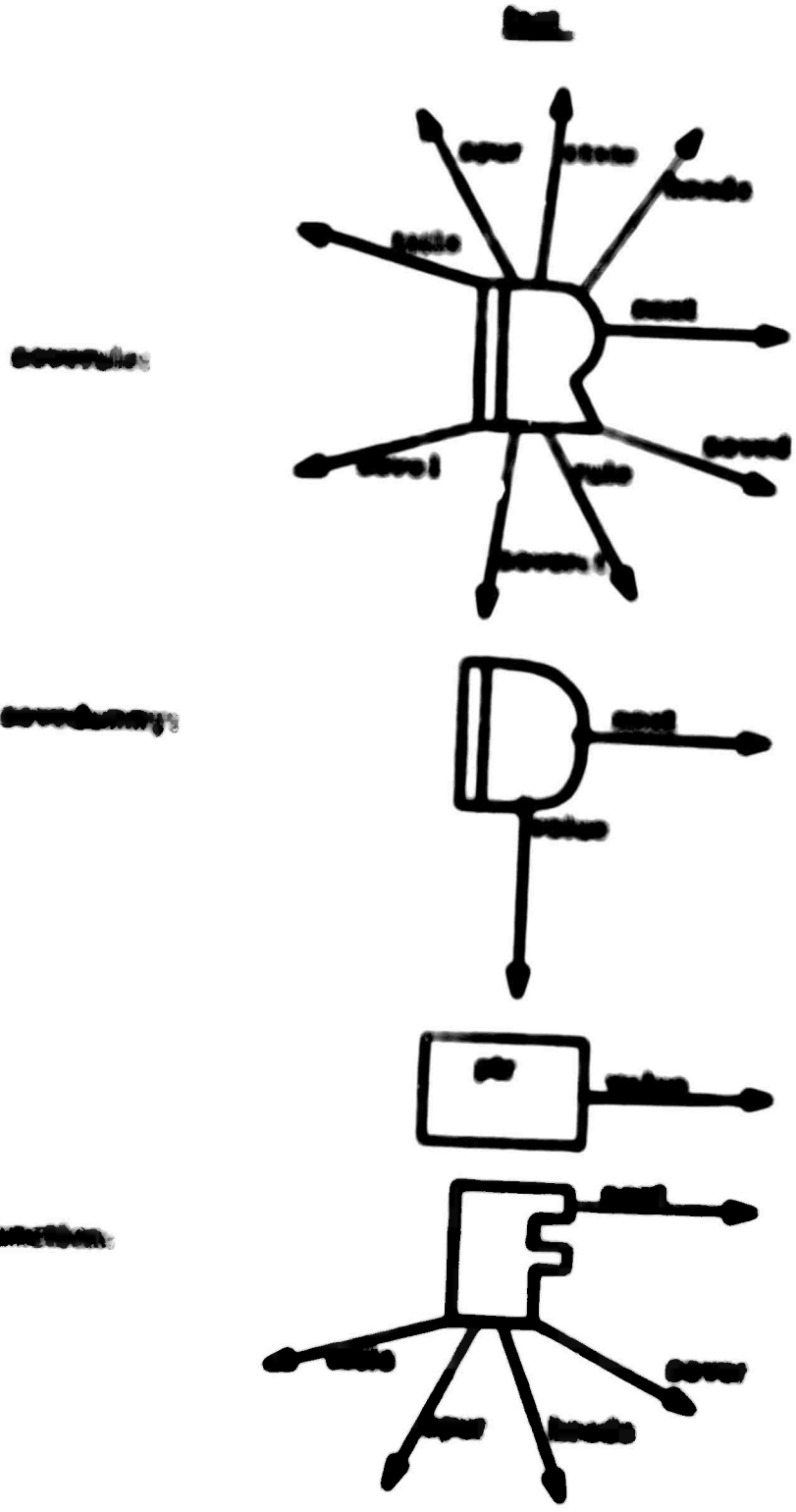
'rule' to the latest value of 'gs out'. It also sets 'gs next_rule' to the first rule of the function being called and then transfers control to 'rule go'. The evaluator package replaces 'rule go' with three rules on pages 'go.1' and 'go.2'. The effect is to push a 'function' node on the evaluation call stack if the evaluation switch is ON. It then saves the table, spur, and bands of the call being made on the 'function' node. It initializes the 'saved' list of the 'function' node to be empty. Finally, it transfers control to the user function.

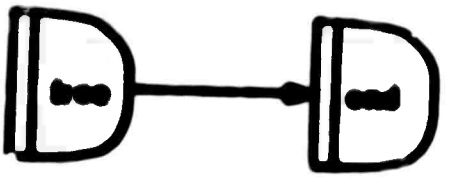
When the ABSET/G interpreter attempts to begin to interpret a potentially interpreted rule, it transfers to 'rule body' after setting 'gs next_rule' to the 'rule' in question. Unless a program happens with the 'stack' hack or a user tries to restart interpretation of a program which formerly stopped in the midst of interpreting rules, this situation can come about only by a rule which includes an active function call being re-entered. This is evidently what happens on recursively called functions. The built-in 'rule body' does nothing but branch to 'rule enter', but the evaluator package replaces it with three rules on pages 'bp.1', 'bp.2', and 'bp.3'. When a rule is re-entered its volatile contents are saved in a 'saved=' node on the 'saved' list of the 'function' node on the top of the evaluator call stack. This includes the values pointed to by the following links of the 'rule' node: 'table', 'spur', 'bands', 'state', 'saved' and 'saved='. In addition, the 'rule' being saved is itself saved on the 'rule' link of the 'saved=' node.

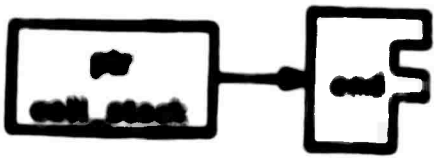
Furthermore, the dummies of the re-entered rule must be saved; this is done on the 'saved' list of the 'saved=' node being used. A list of 'saved=dummy's saves all of the dummies of the re-entered rule. These are found by scanning for the 'testeny's of the 'frame' list of the rule which represents the binding of a dummy; these are found only on the 'test' list of a 'testeny'. See the 'scan_testeny' function on pages 'sn.2' and 'sn.3'. It is actually used both for saving and restoring dummies. Its second local argument indicates the name of a function to be called once with each 'testeny' as argument which is a dummy. The first local argument of 'scan_testeny' is the first element of a 'frame' list of 'testeny's. During saving 'scan_testeny' calls 'save_psy' given on page 'sv.2'; during restoration it calls 'restore_psy' given on page 'rv.2'.

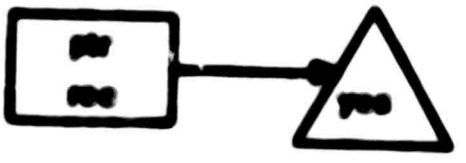
When the [illegible] interpreter processes a user function which does so as a result of interpreting 'rule step'. The user, however, is told to branch to 'rule out' to return from a function. The built-in 'rule out' does nothing but branch to 'rule step', but the execution package replaces it with its rules on pages '6.5', '6.6', and '6.8'. The processing of a function return essentially pops the recursive call stack. As it does so it also restores all saved re-entered rules which were interpreted during the execution of the function in question. It does this by scanning down the 'saved' list of the 'function' node and performing a restoration for every 'saved=rule' it finds.

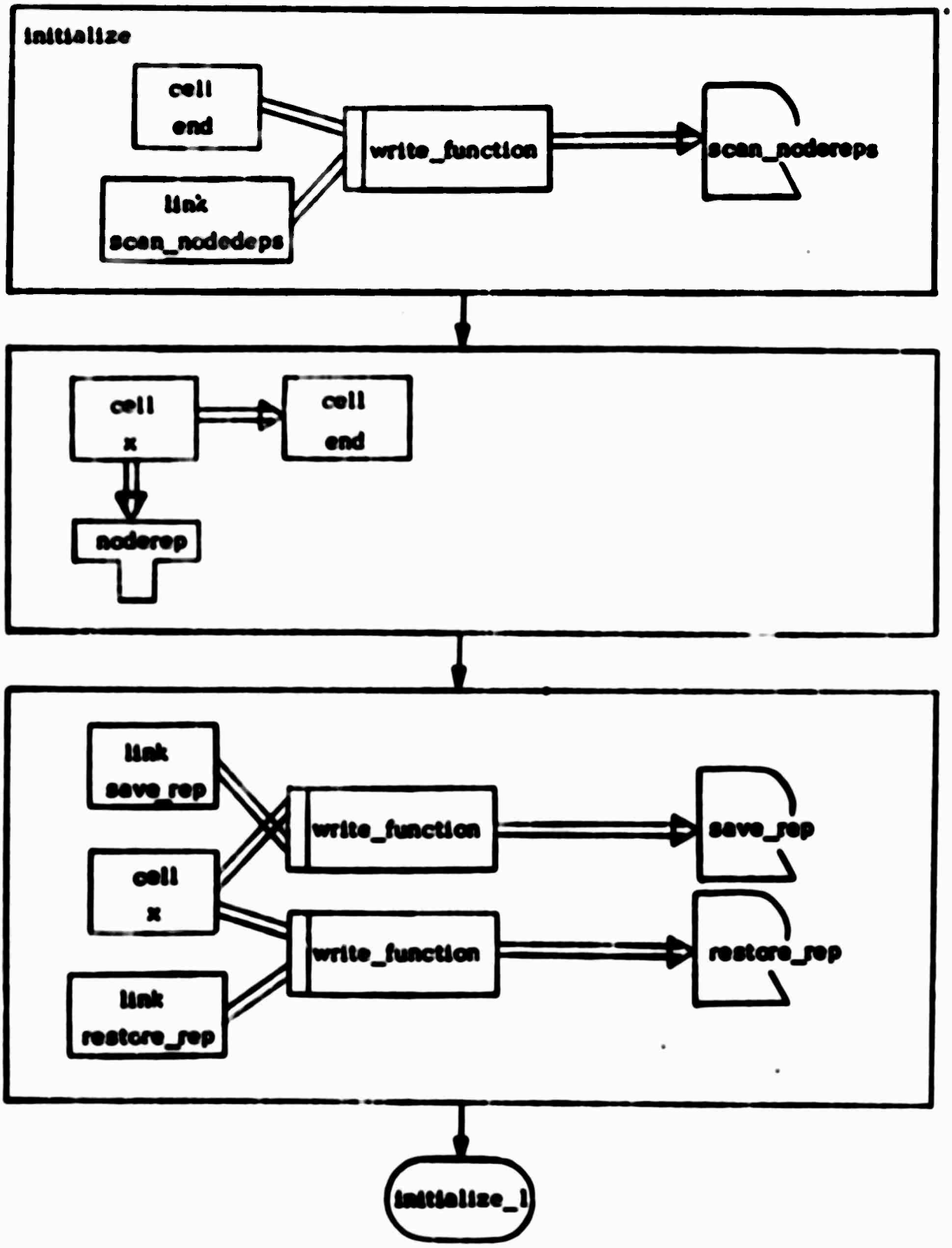
The remaining functions of the 'hex' program are general utility functions for maintaining hex lists and passing arguments and results. The reader may, want to study the 'i_init_j' and 'return_j_host_j' functions on pages '6.6' and '6.7' (the last two pages of the listing). The need for these 'strange' functions has prompted us to want to reconsider the design of argument and result passage in the [illegible] language.

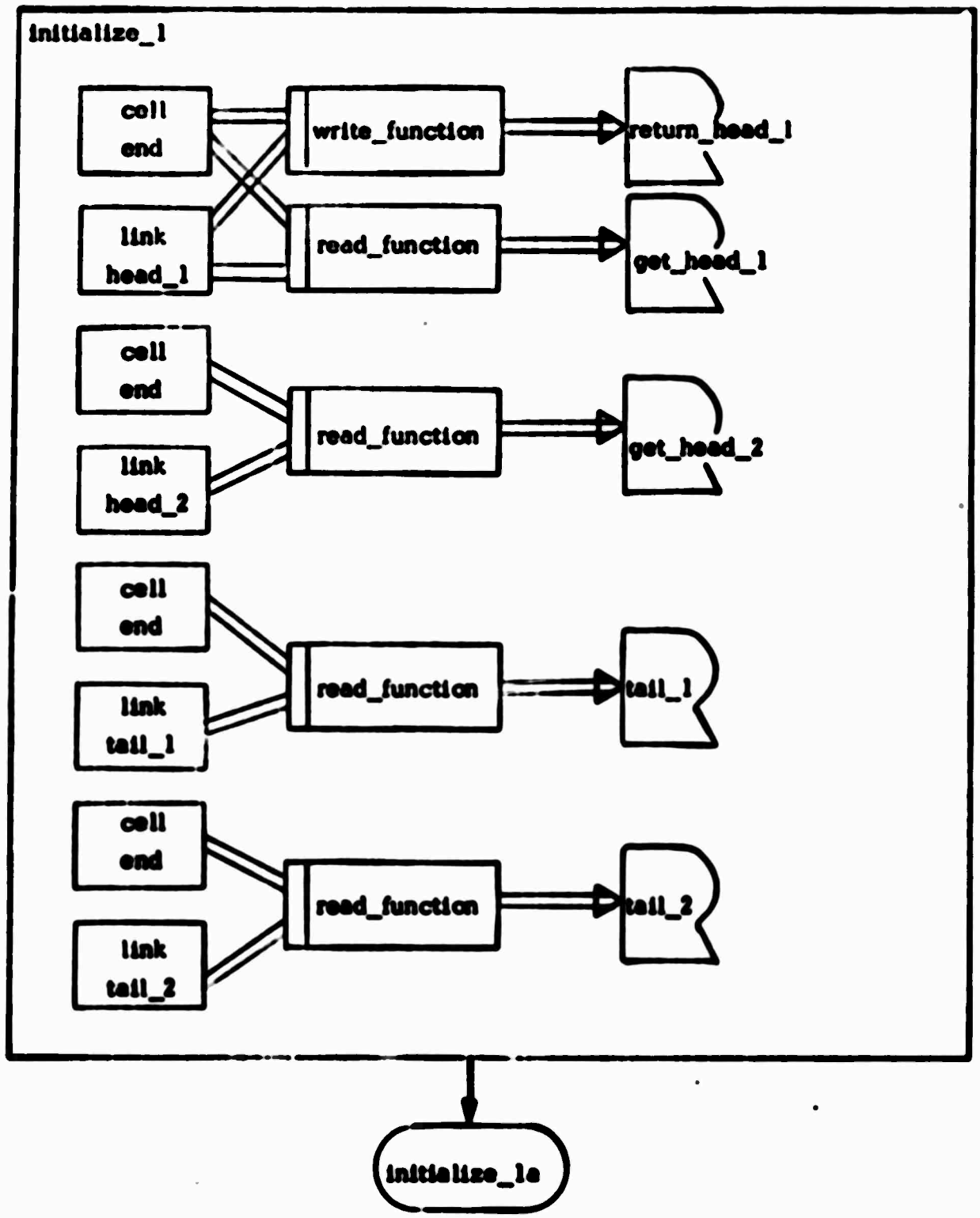


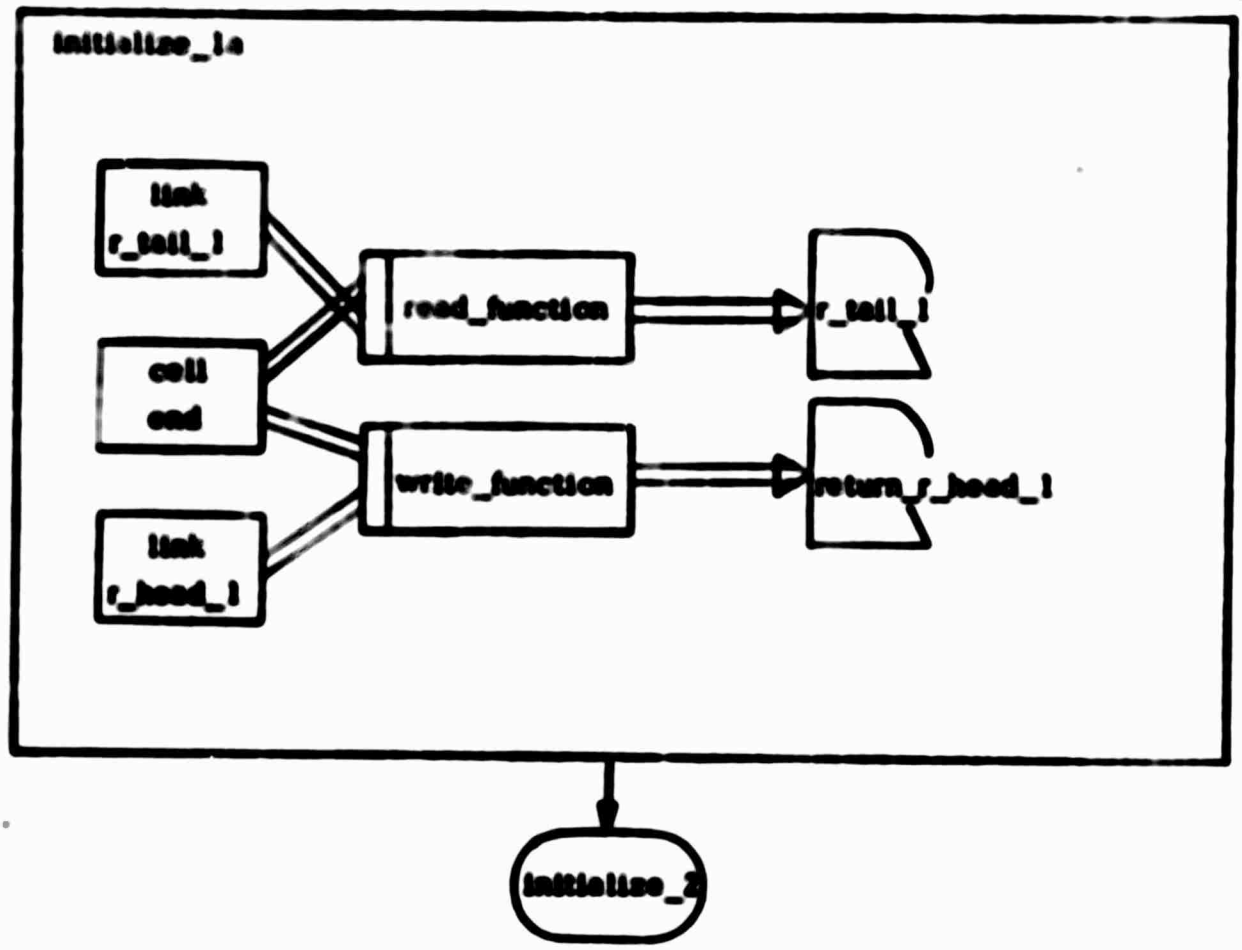


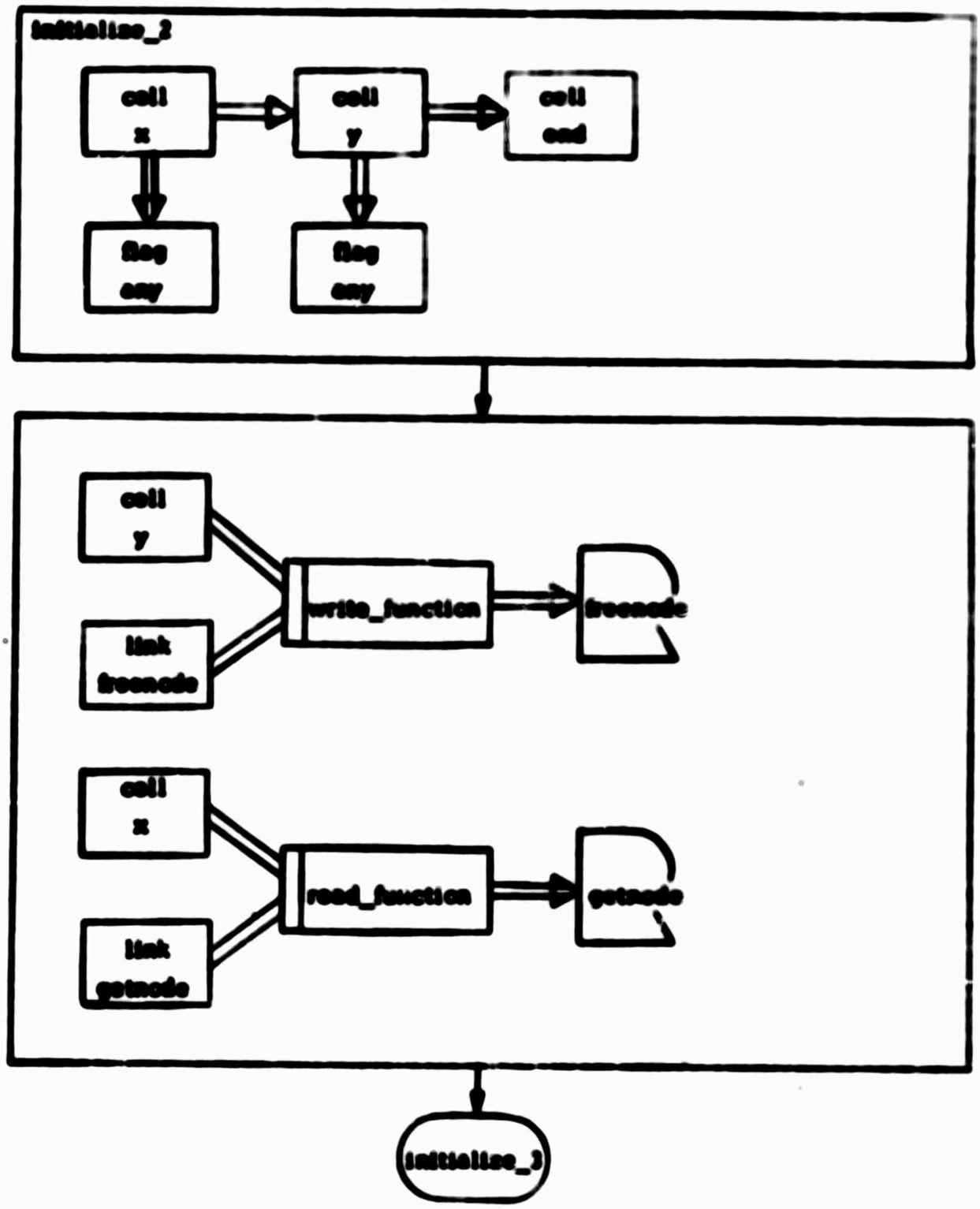


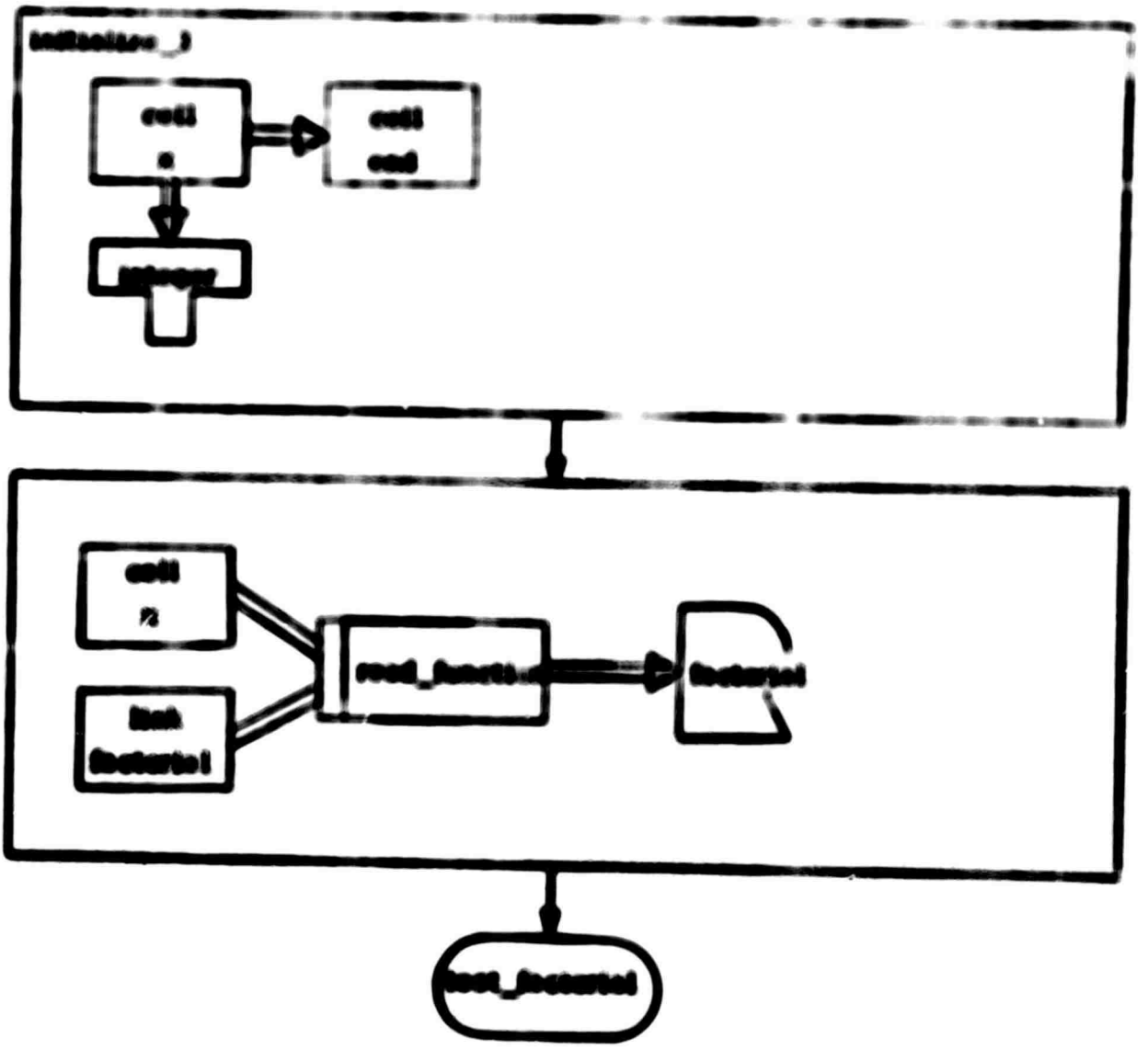


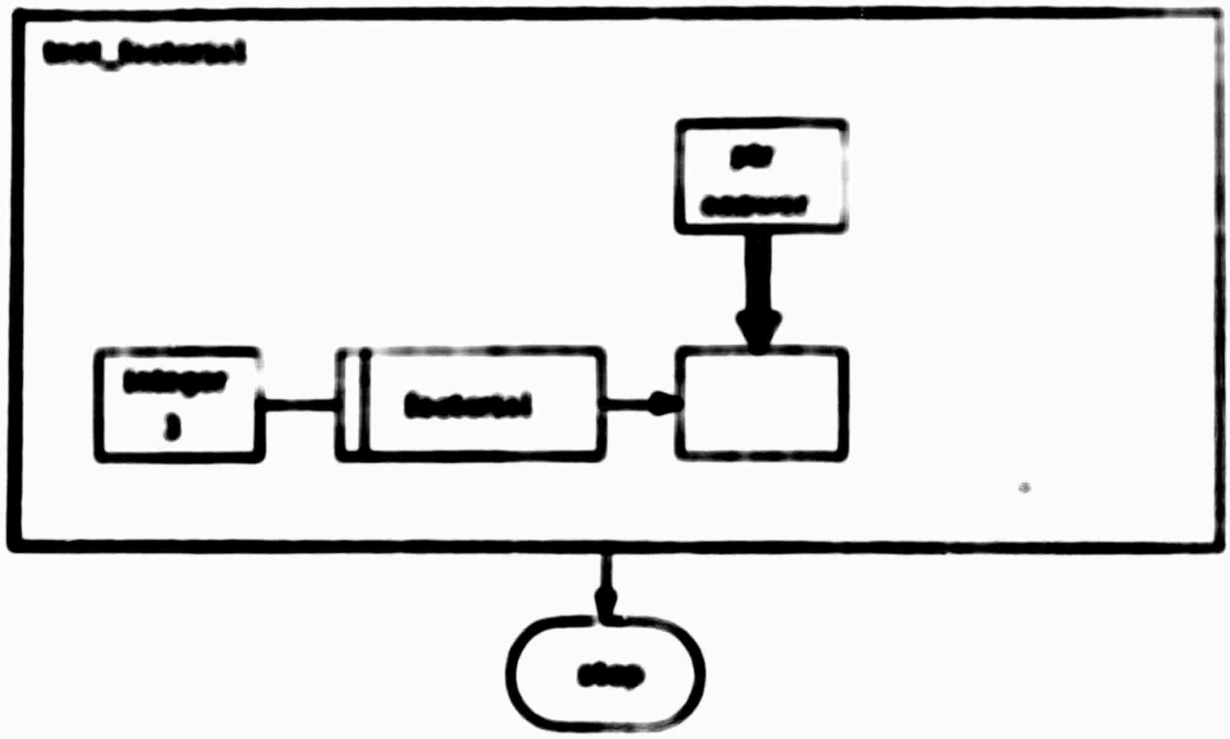

initialize
cell
end
link
scan_nodedeps
write_function
scan_nodereps
cell
x
cell
end
noderep
link
save_rep
cell
x
link
restore_rep
write_function
write_function
save_rep
restore_rep
initialize_l

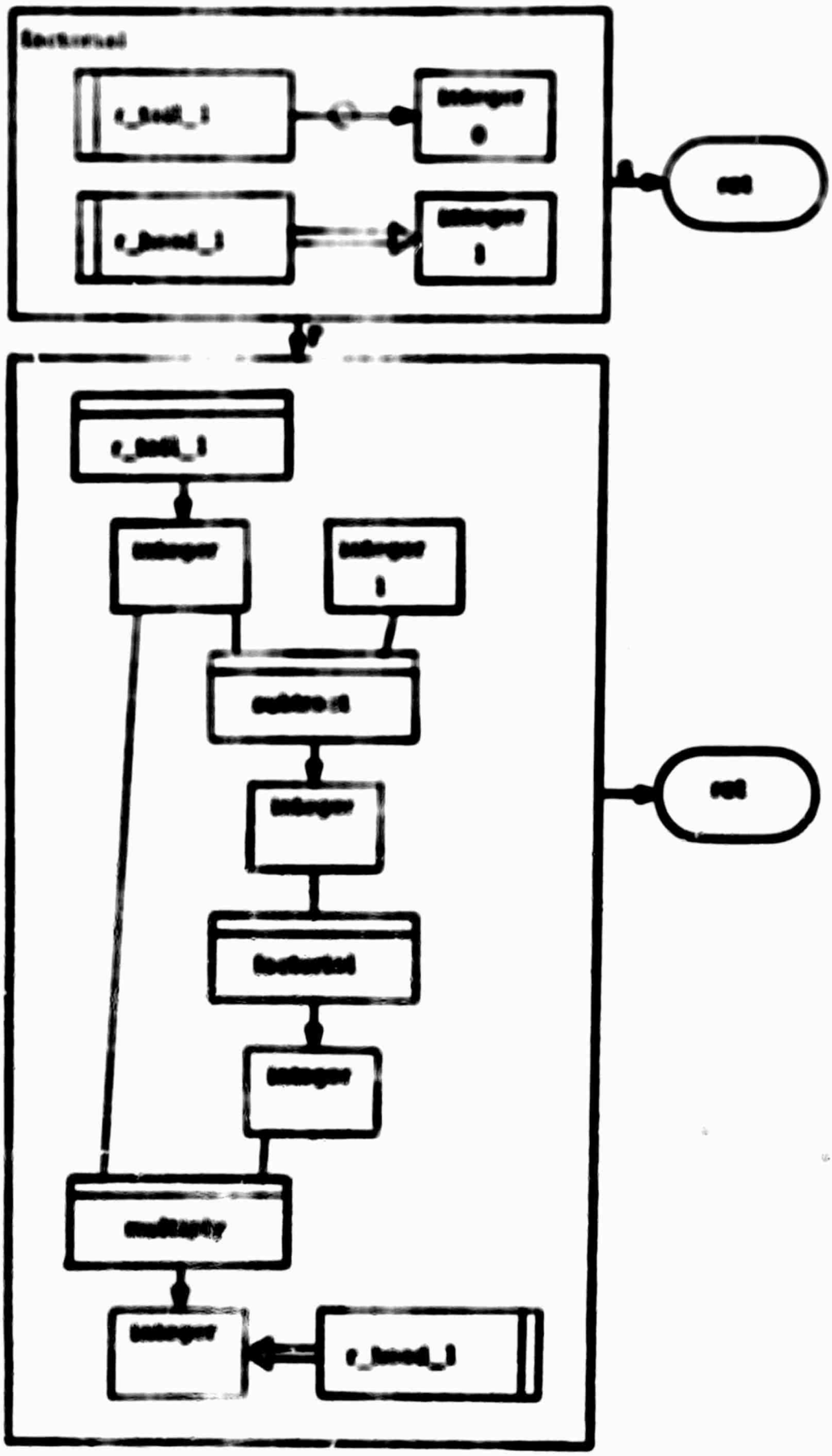

initialize_1
call end
link head_1
write_function
read_function
return_head_1
get_head_1
call end
link head_2
read_function
get_head_2
call end
link tail_1
read_function
tail_1
call end
link tail_2
read_function
tail_2
initialize_1e

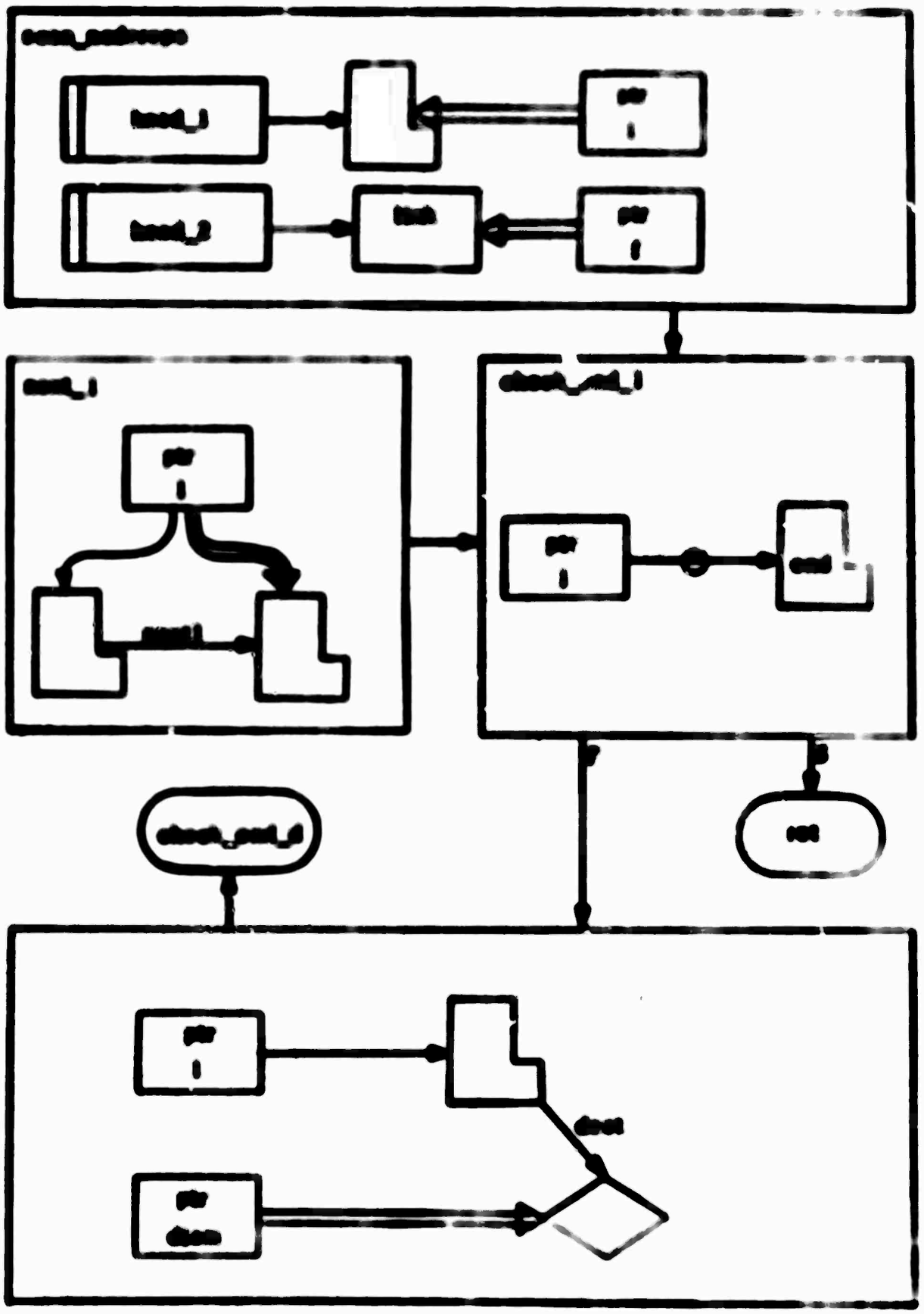

initialize_1a
link
r_tail_1
call
end
link
r_head_1
read_function
write_function
r_tail_1
return_r_head_1
initialize_2

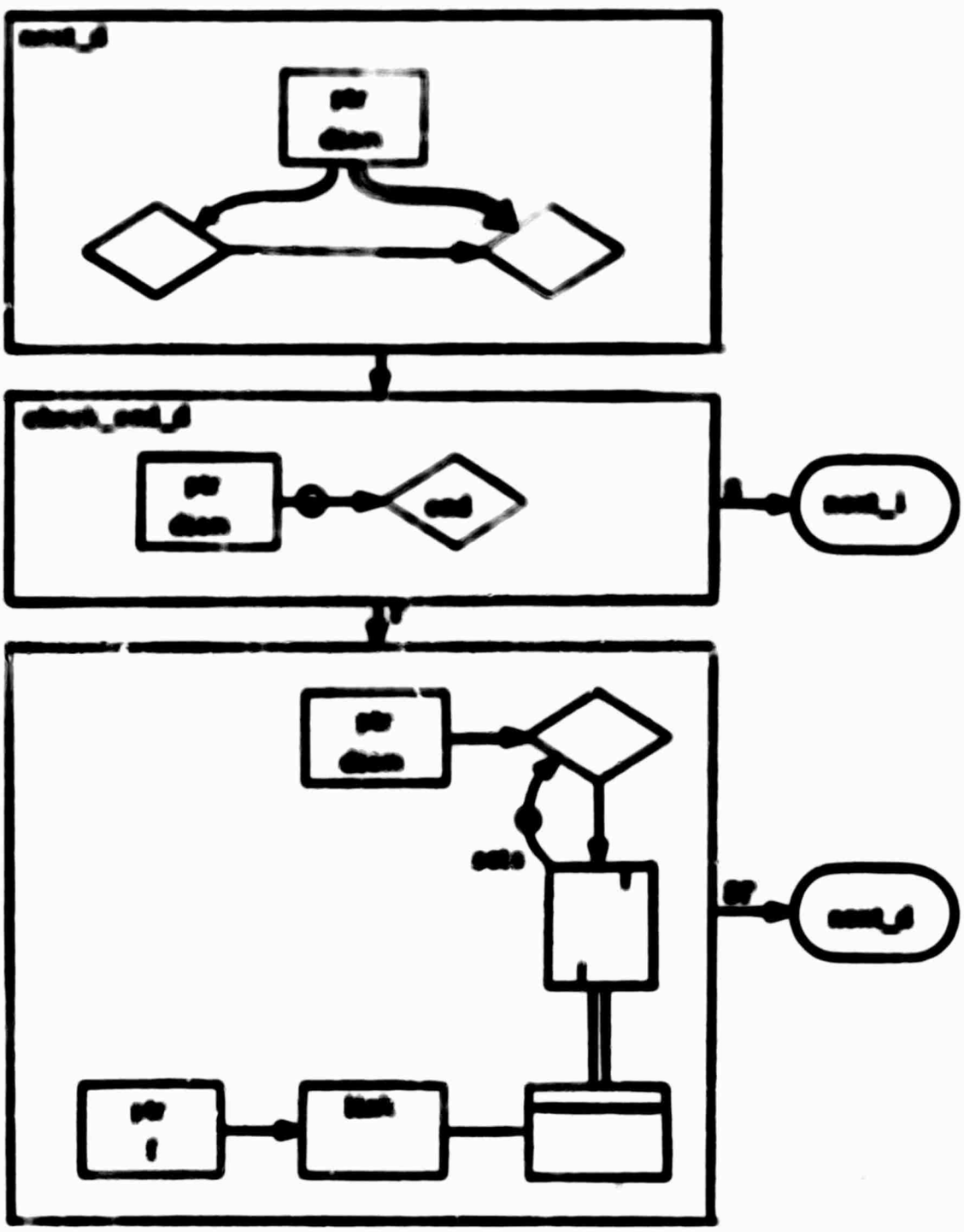

initialize_2
call
x
call
y
call
end
flag
any
flag
any
call
y
link
freenode
write_function
freenode
call
x
link
getnode
read_function
getnode
initialize_3

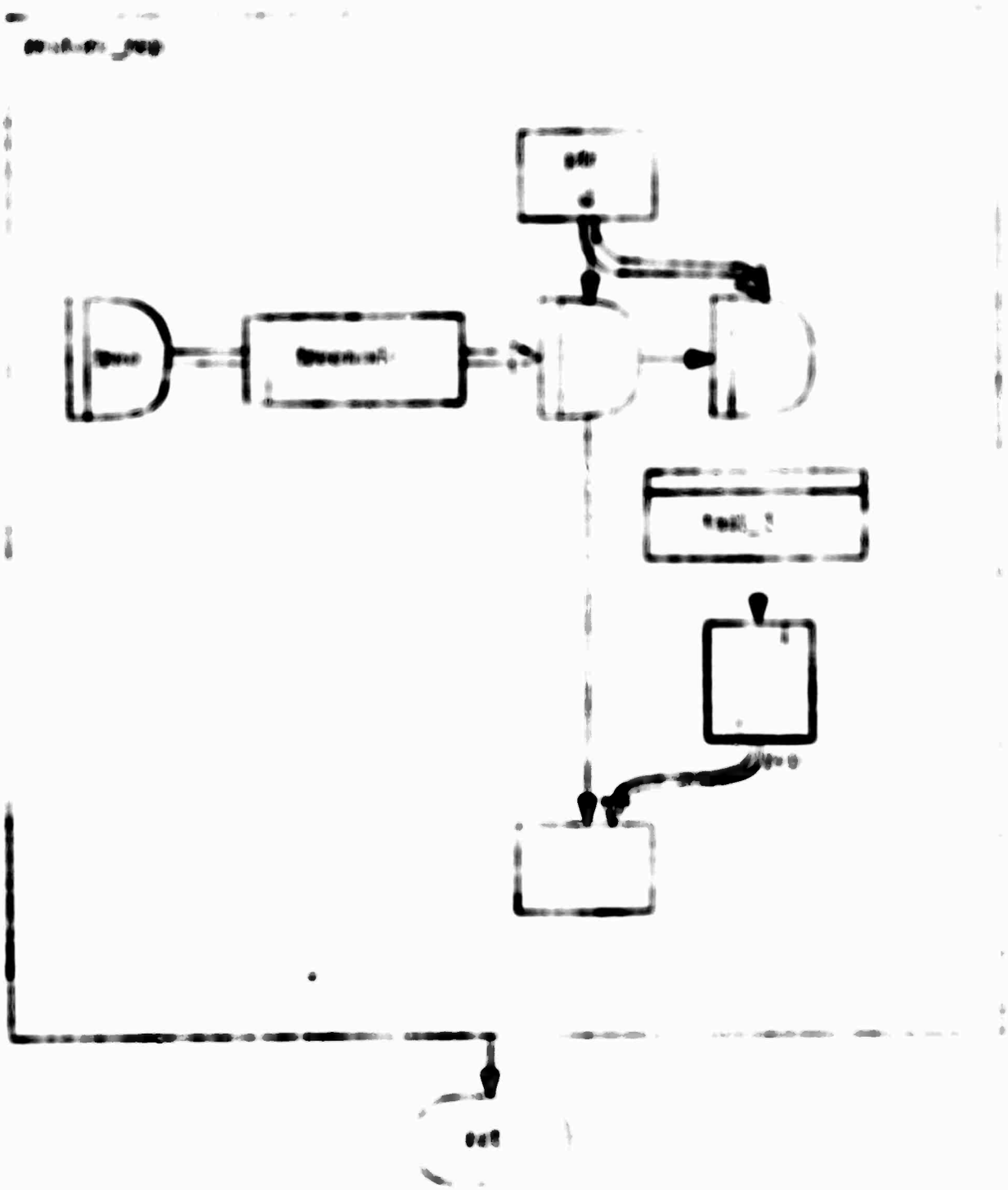


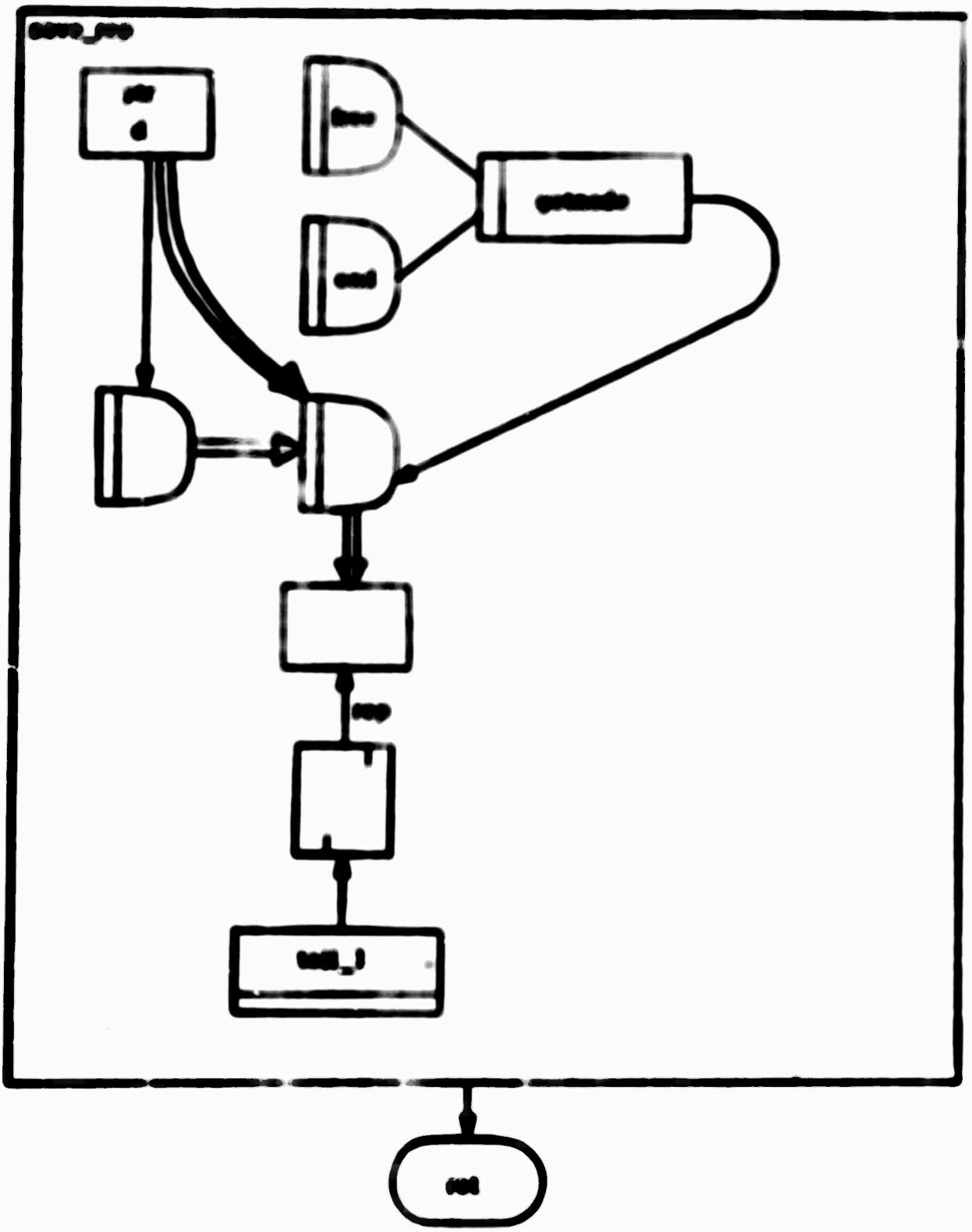


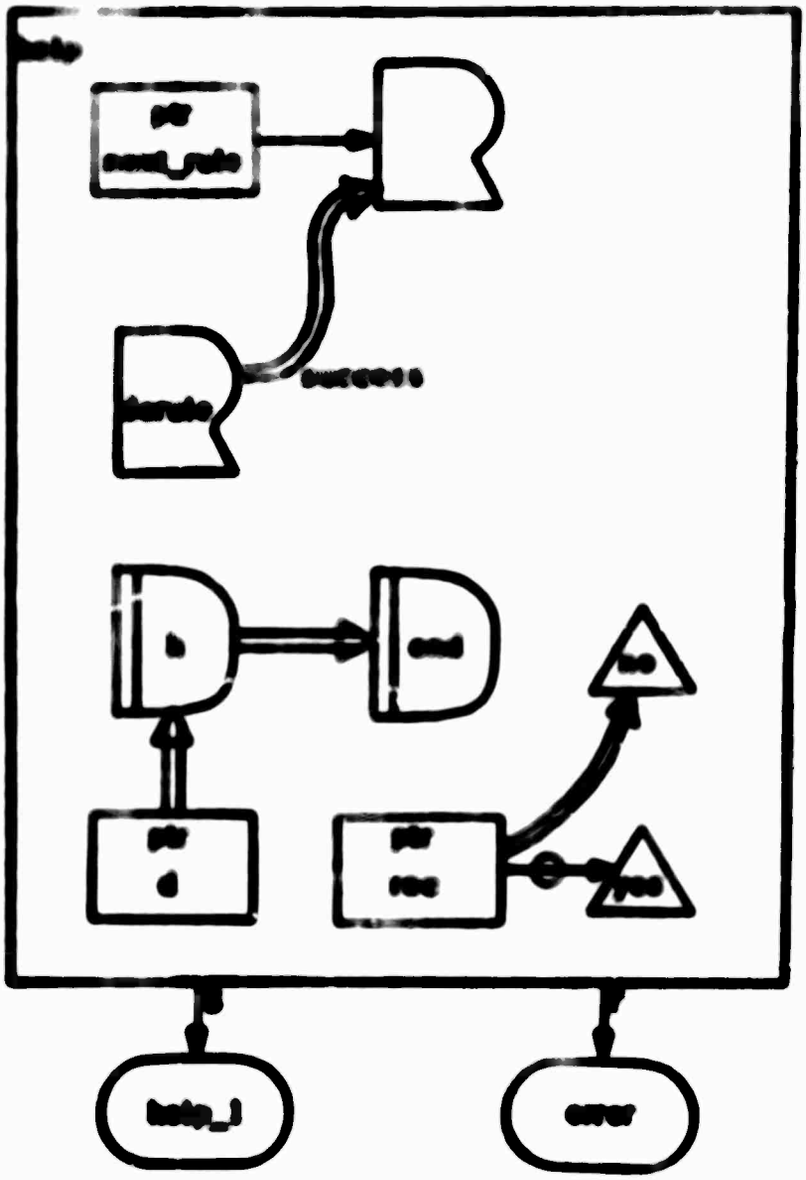


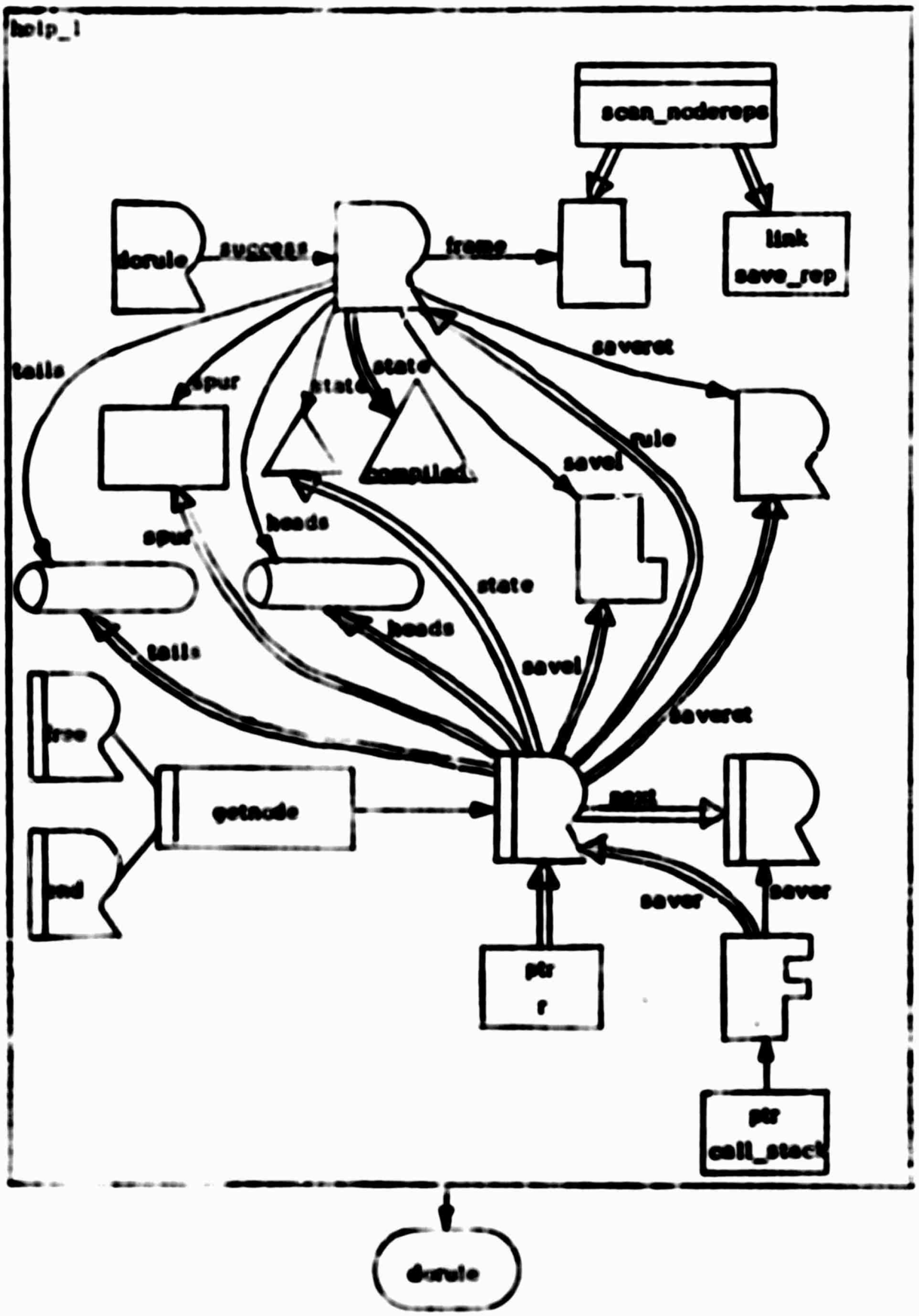


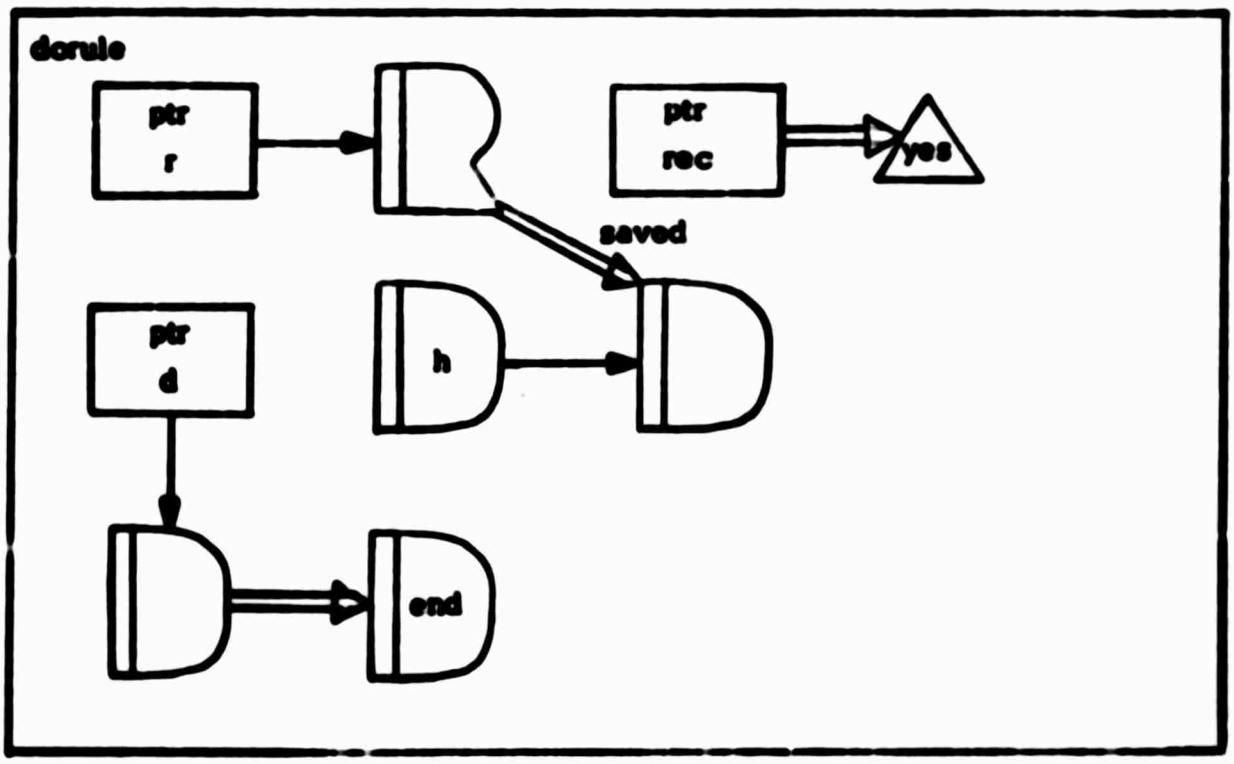


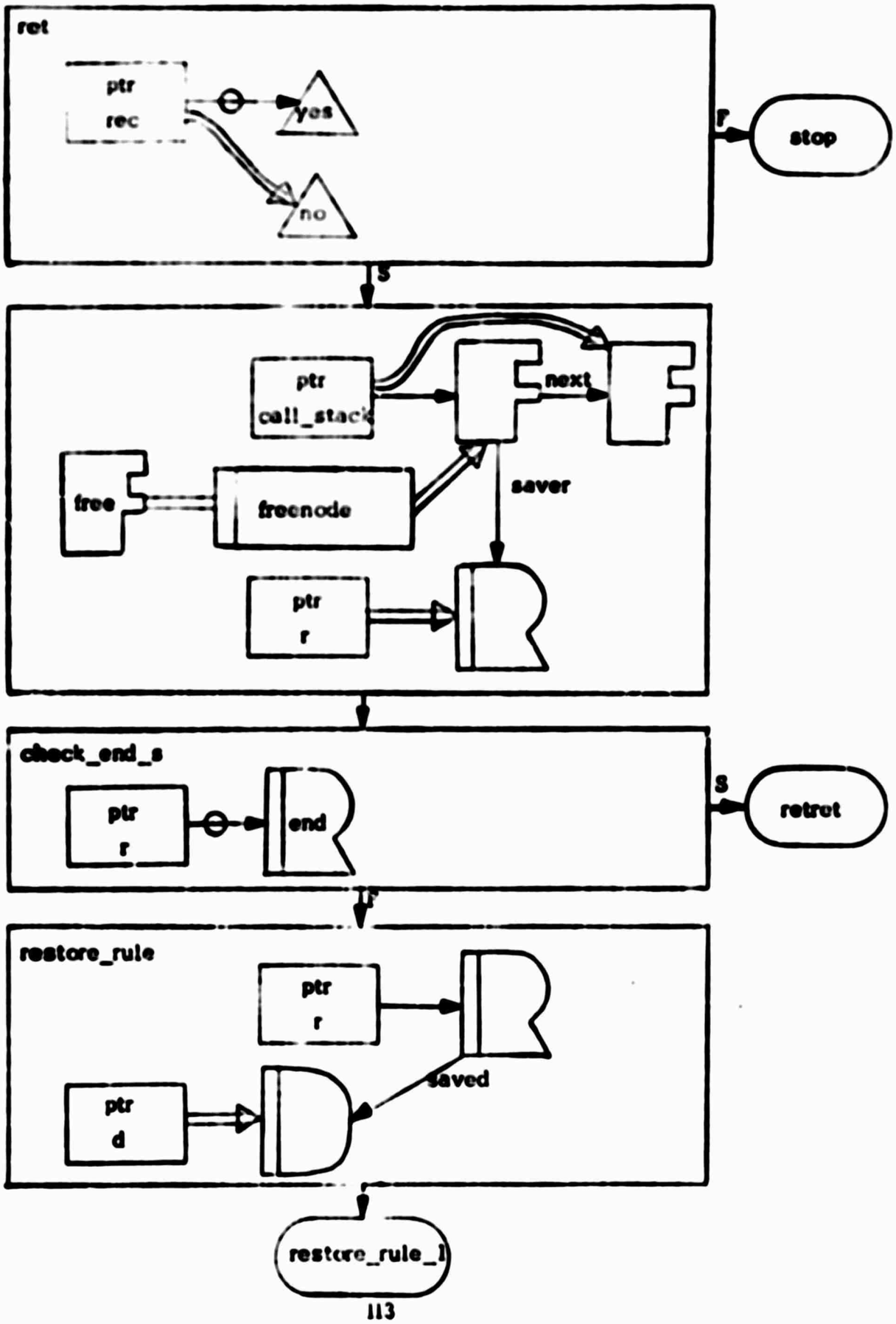


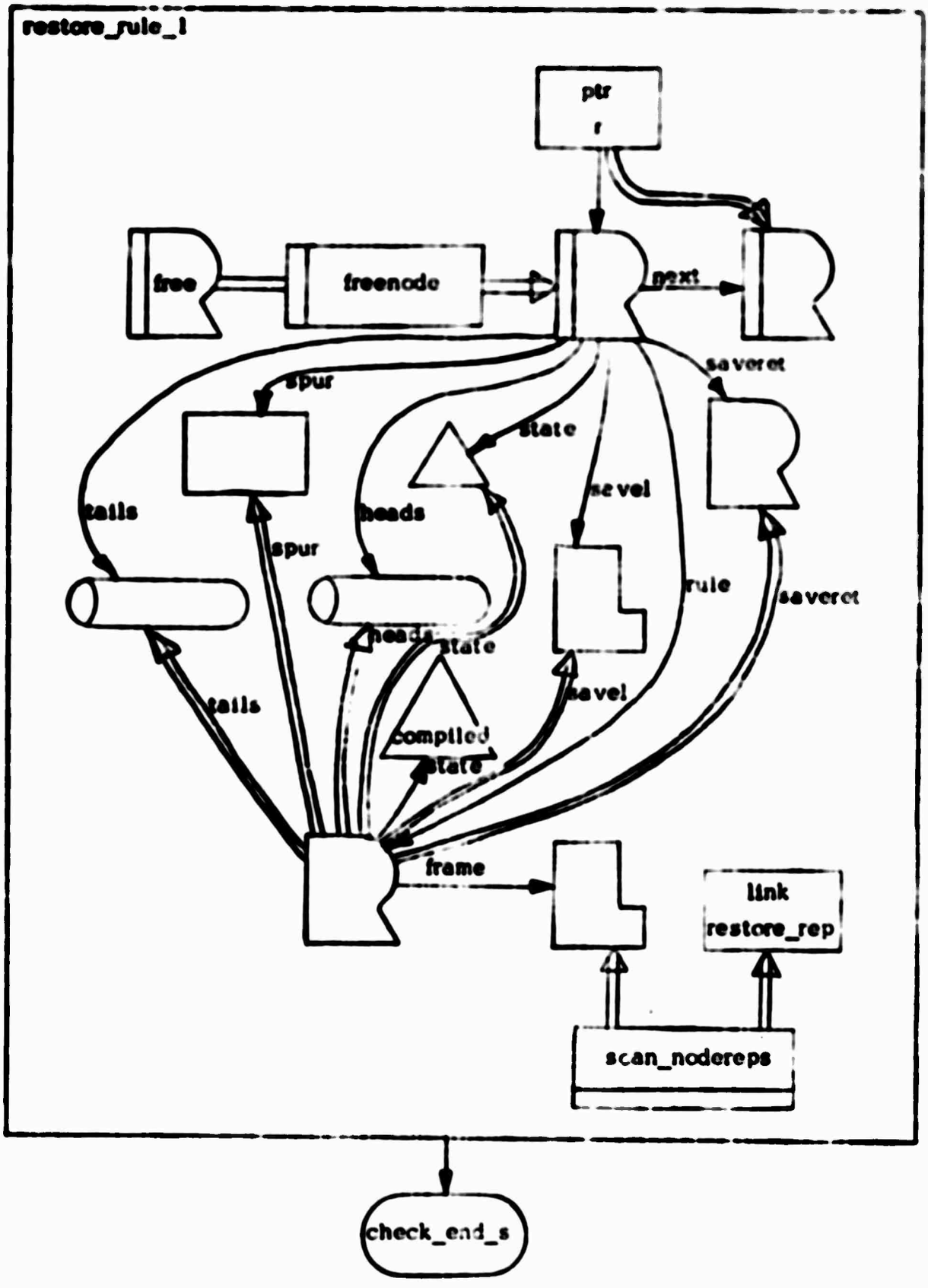


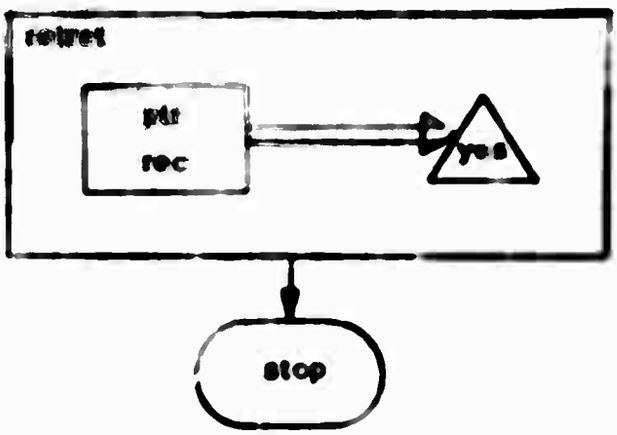


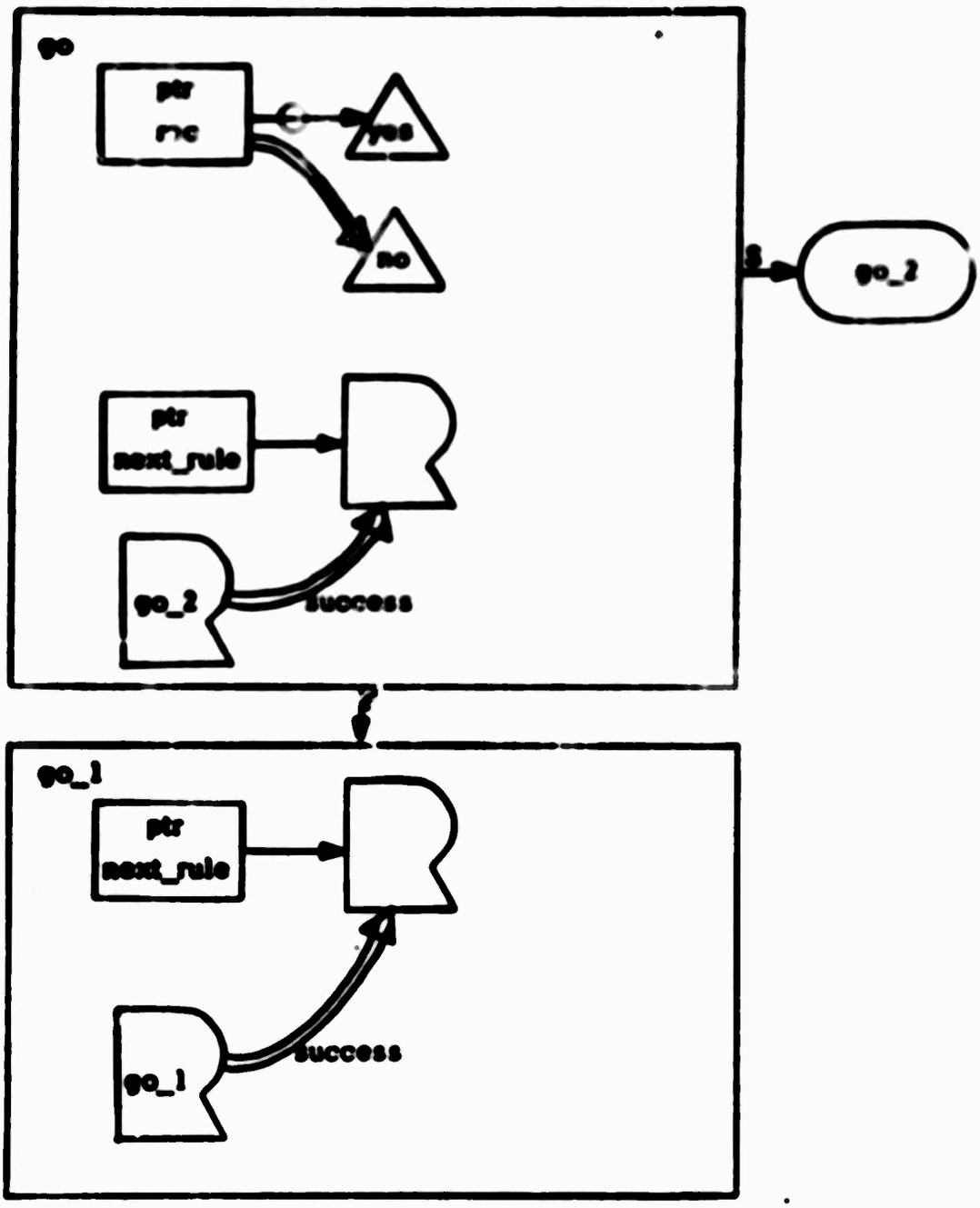

help_1
scan_noderops
link
save_rep
success
frame
tails
spur
state
compiled
rule
savel
heads
saveret
getnode
ptr
r
ptr
call_stack
durule

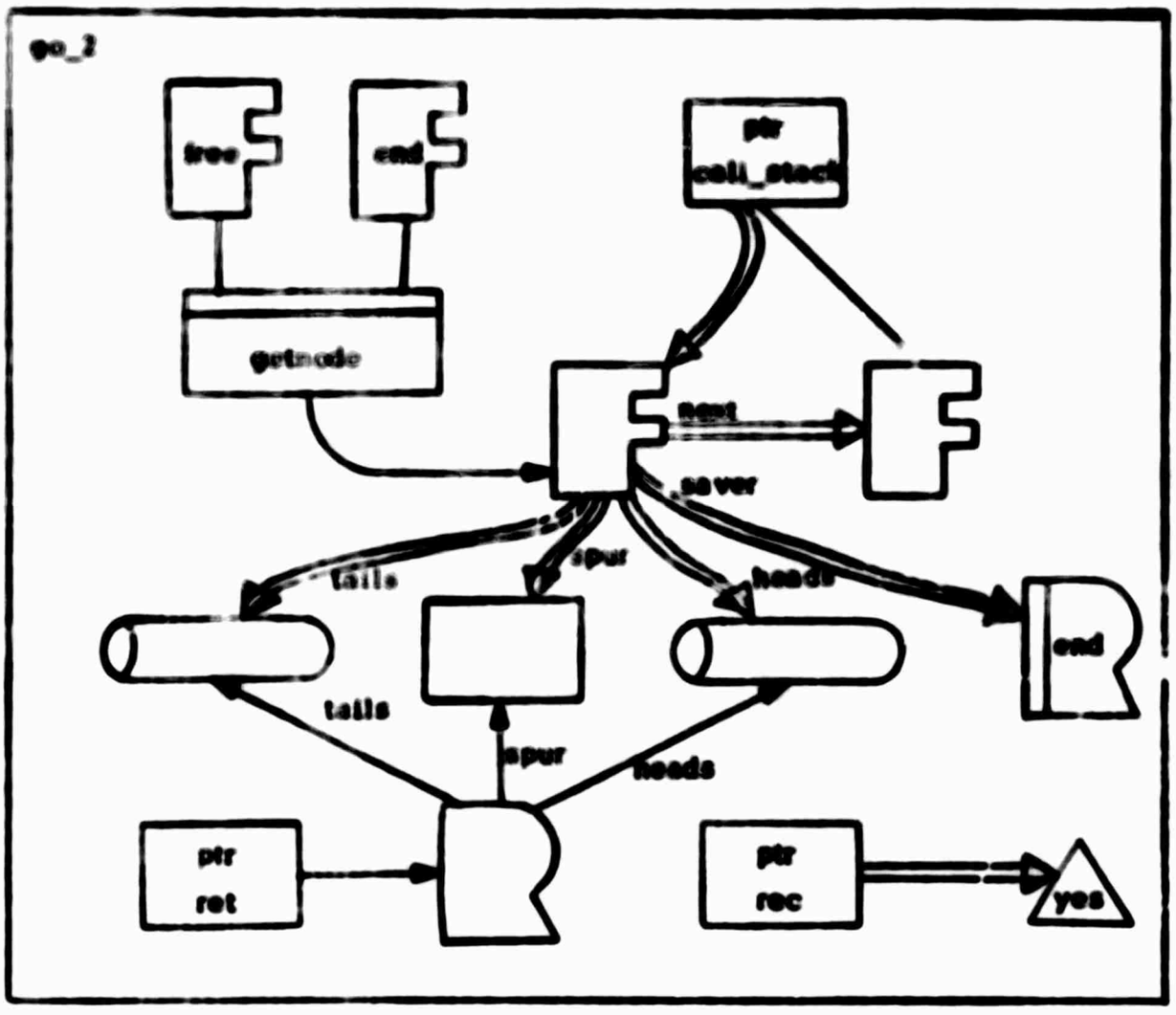

dorule
ptr
r
ptr
rec
yes
saved
ptr
d
h
end

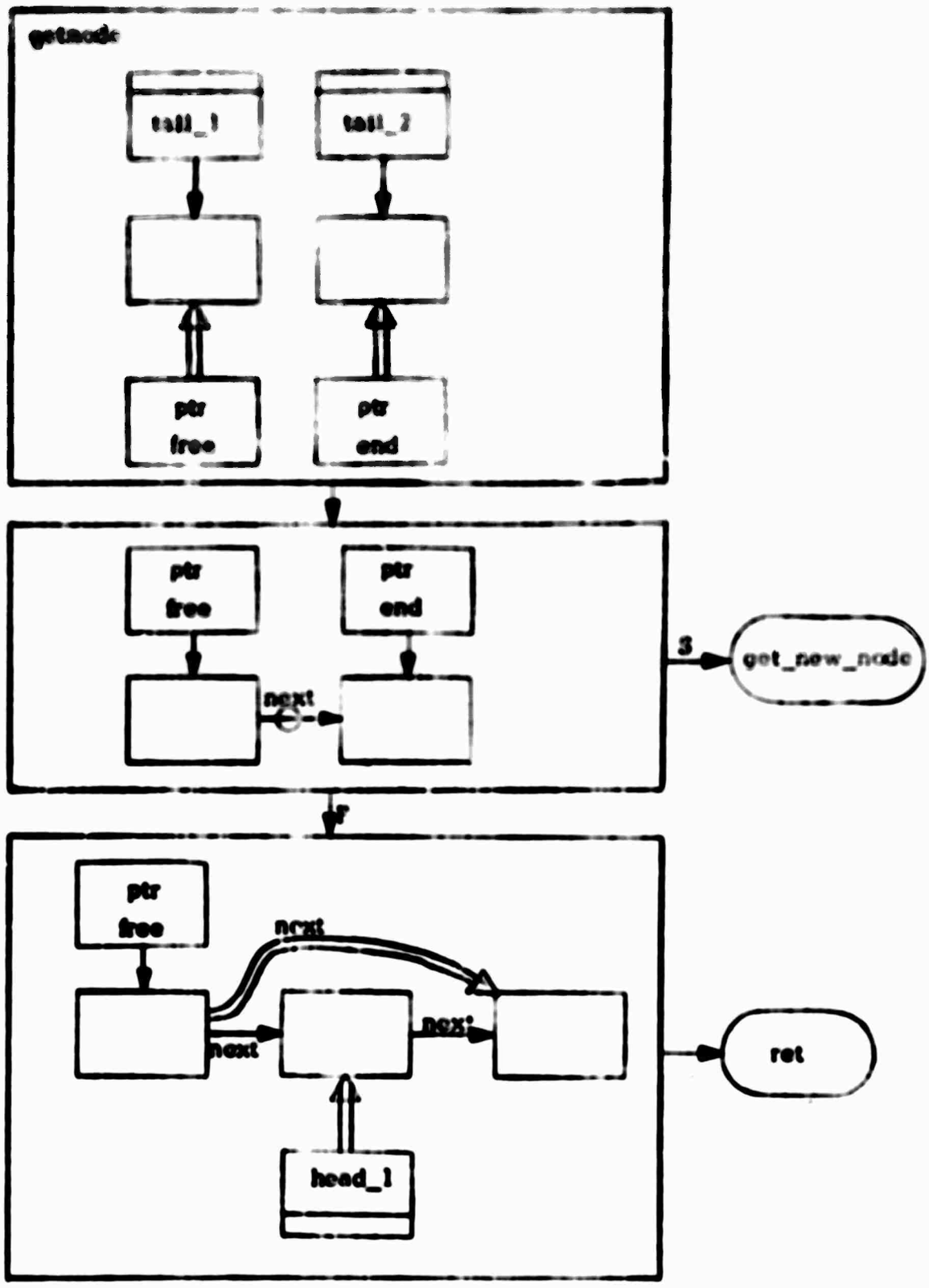

ret
ptr
rec
yes
no
F
stop
S
ptr
call_stack
next
free
freenode
saver
ptr
r
check_end_s
ptr
r
end
S
retret
F
restore_rule
ptr
r
saved
ptr
d
restore_rule_l

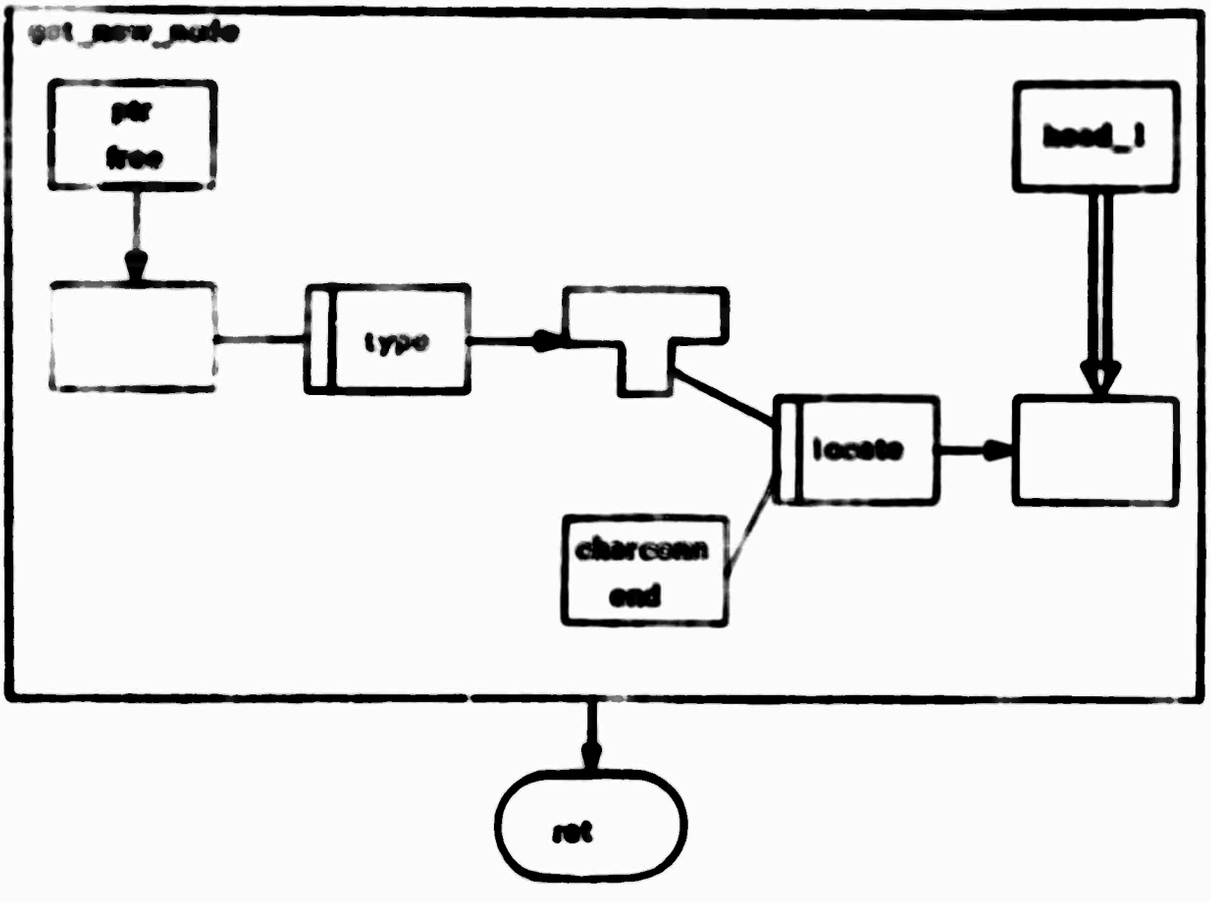

restore_rule_1
ptr
r
free
freenode
next
saveret
spur
state
tails
heads
spur
savel
rule
saveret
heads
state
tails
savel
compiled
state
frame
link
restore_rep
scan_nodereps
check_end_s

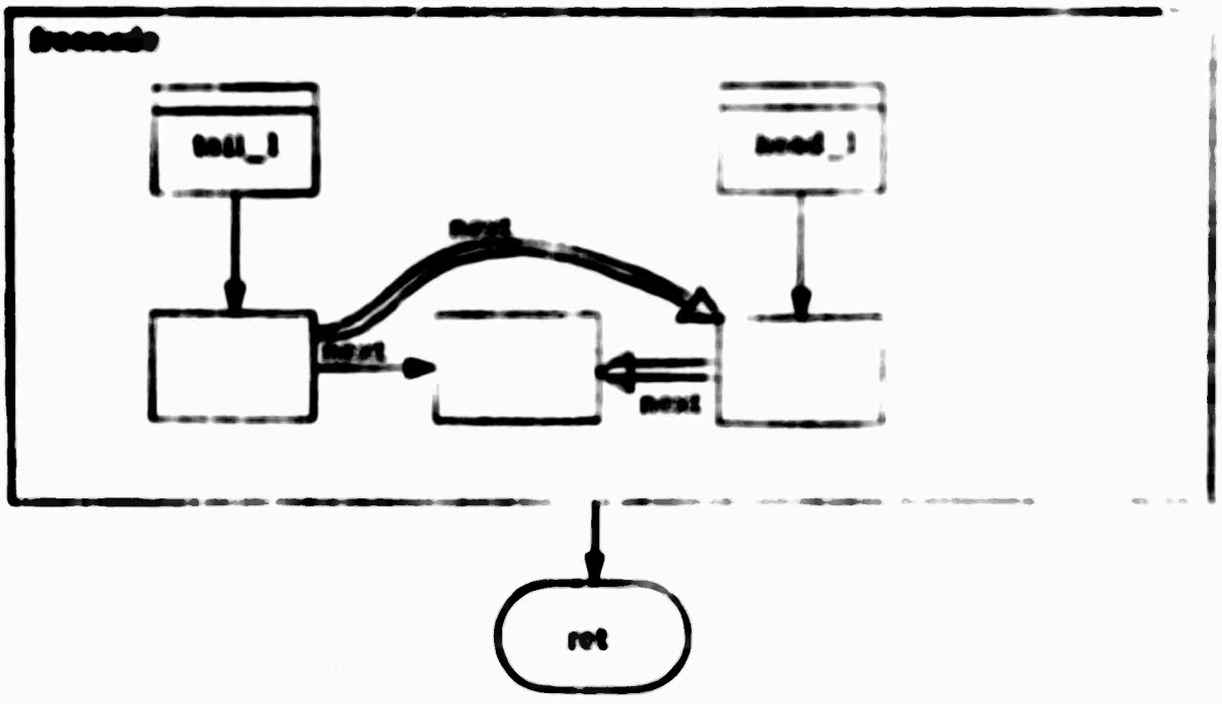

retpret
ptr
rec
yes
stop

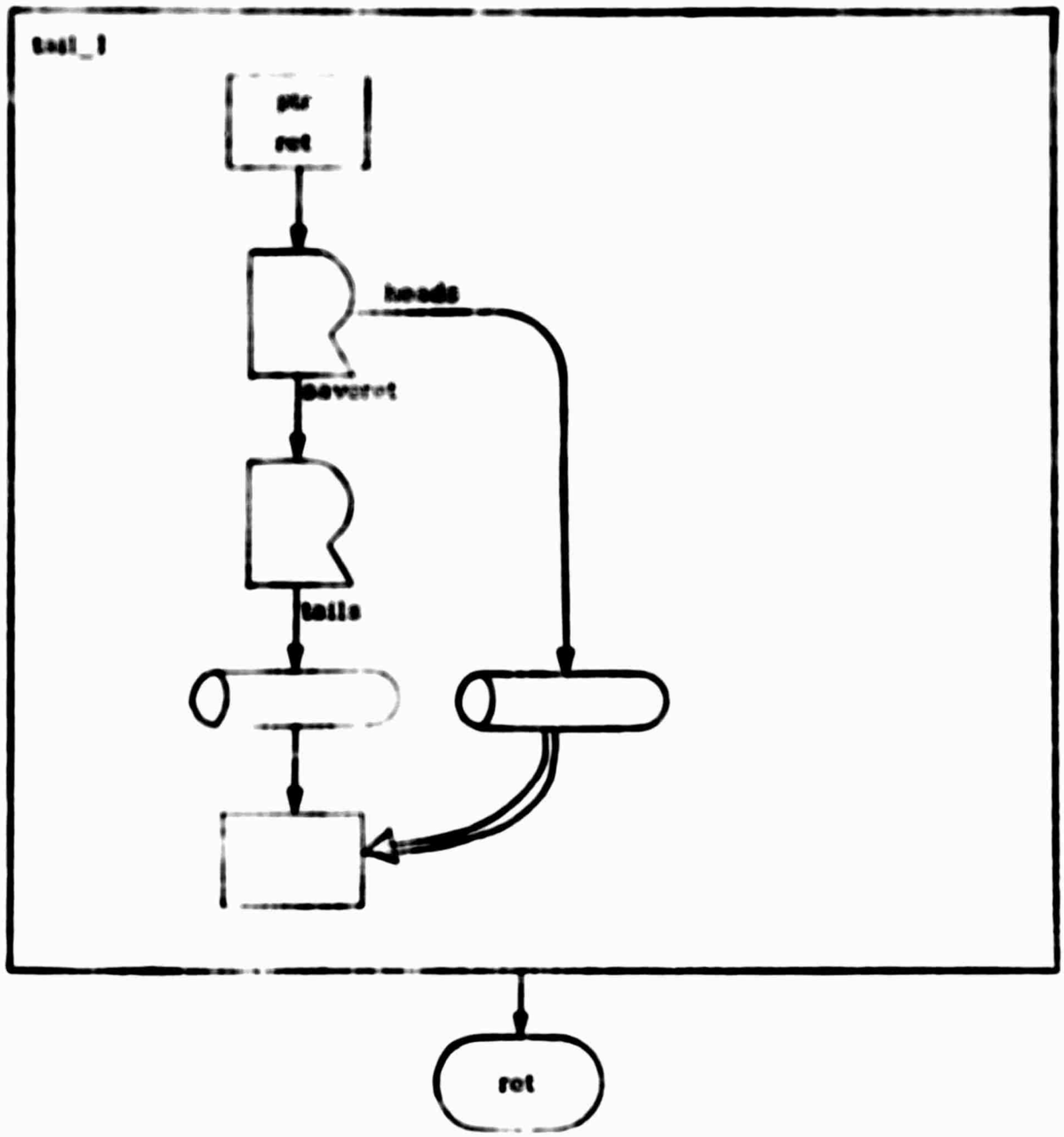

go
ptr
rnc
yes
no
go_2
ptr
next_rule
go_2
success
go_1
ptr
next_rule
go_1
success

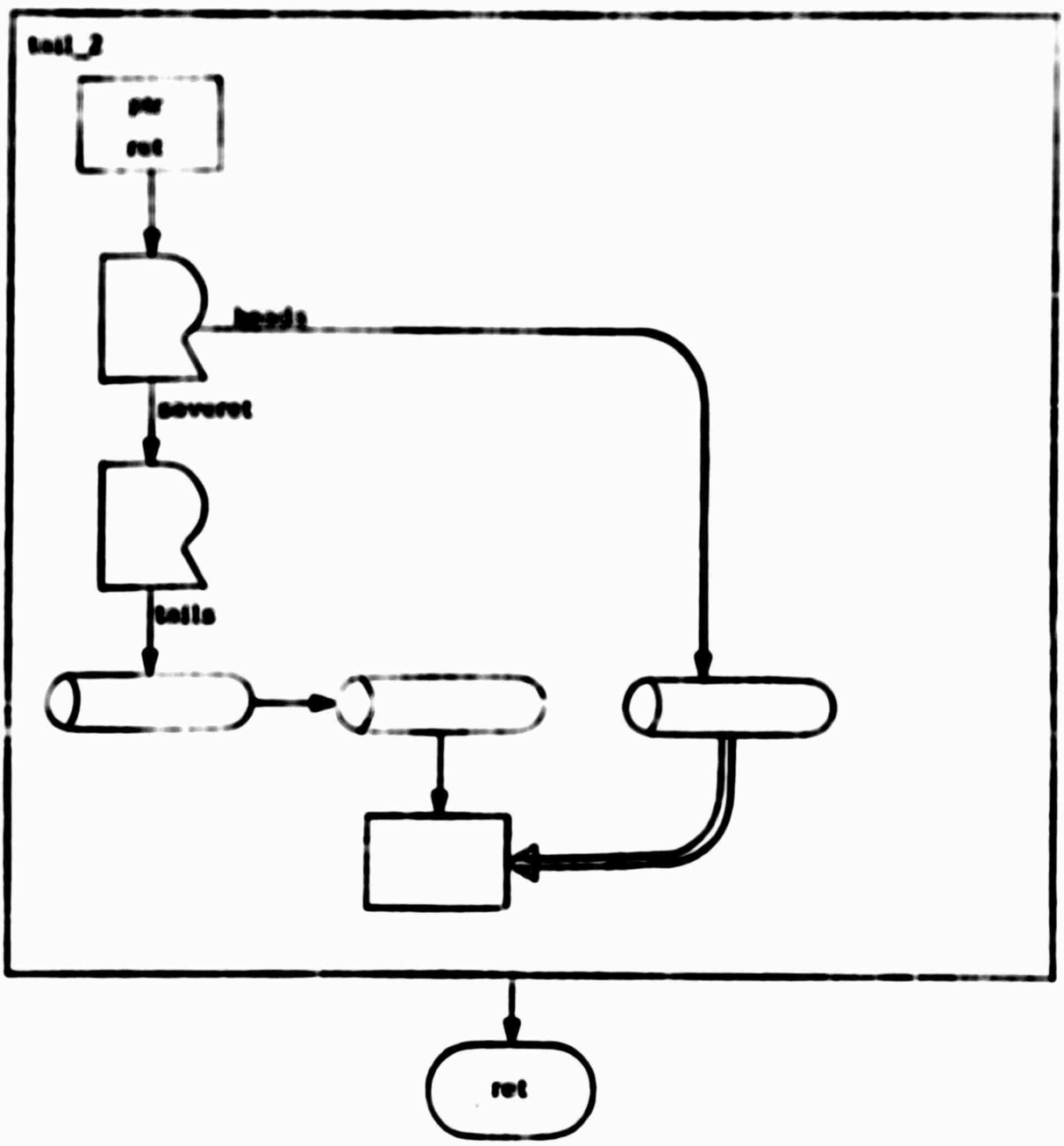

go_2
free
end
ptr
call_stack
getnode
next
prev
tails
spur
heads
end
tails
spur
heads
ptr
ret
ptr
rec
yes

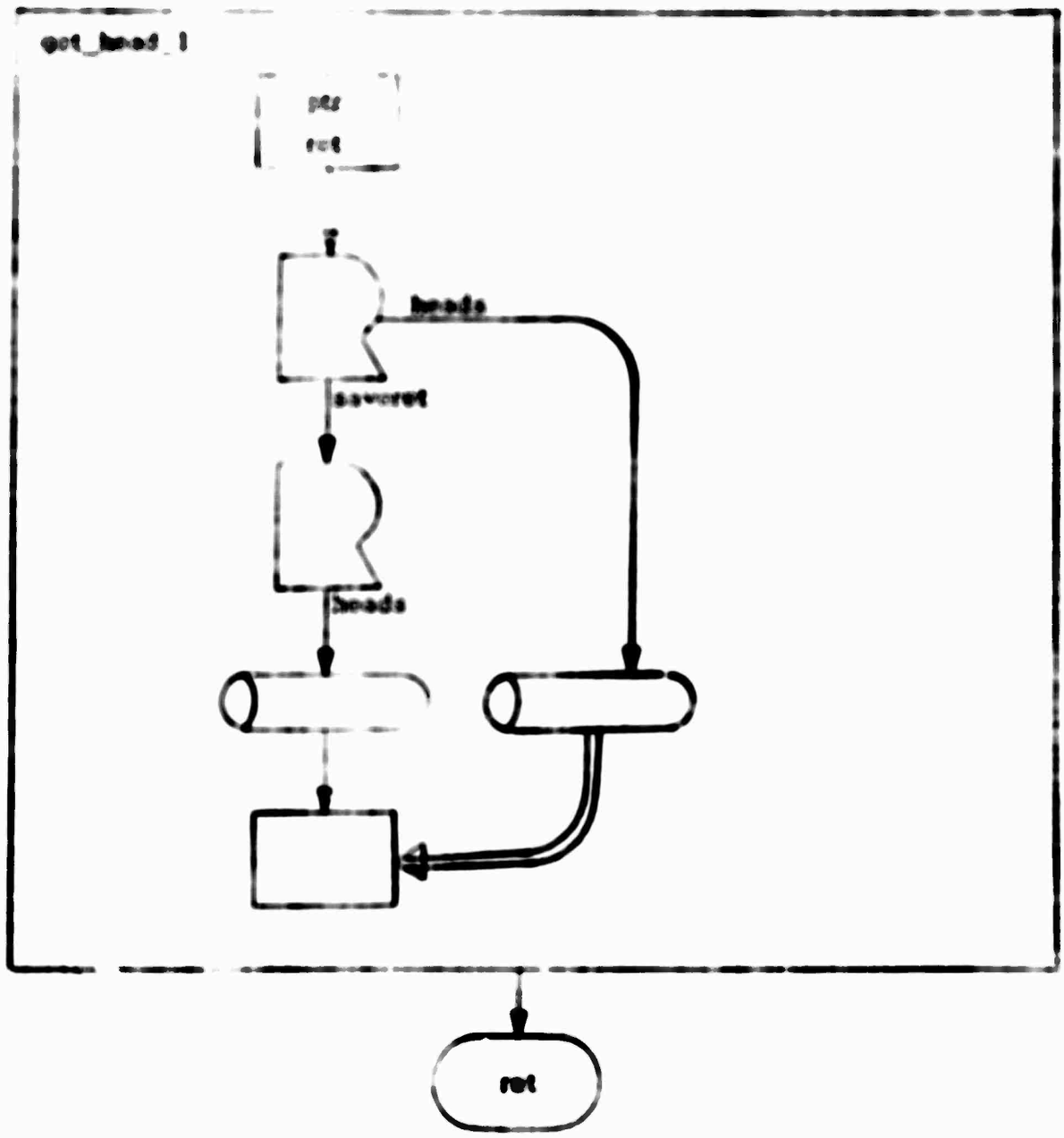

getnode
tail_1
tail_2
ptr
free
ptr
end
ptr
free
ptr
end
next
3
get_new_node
F
ptr
free
next
next
next'
ret
head_1

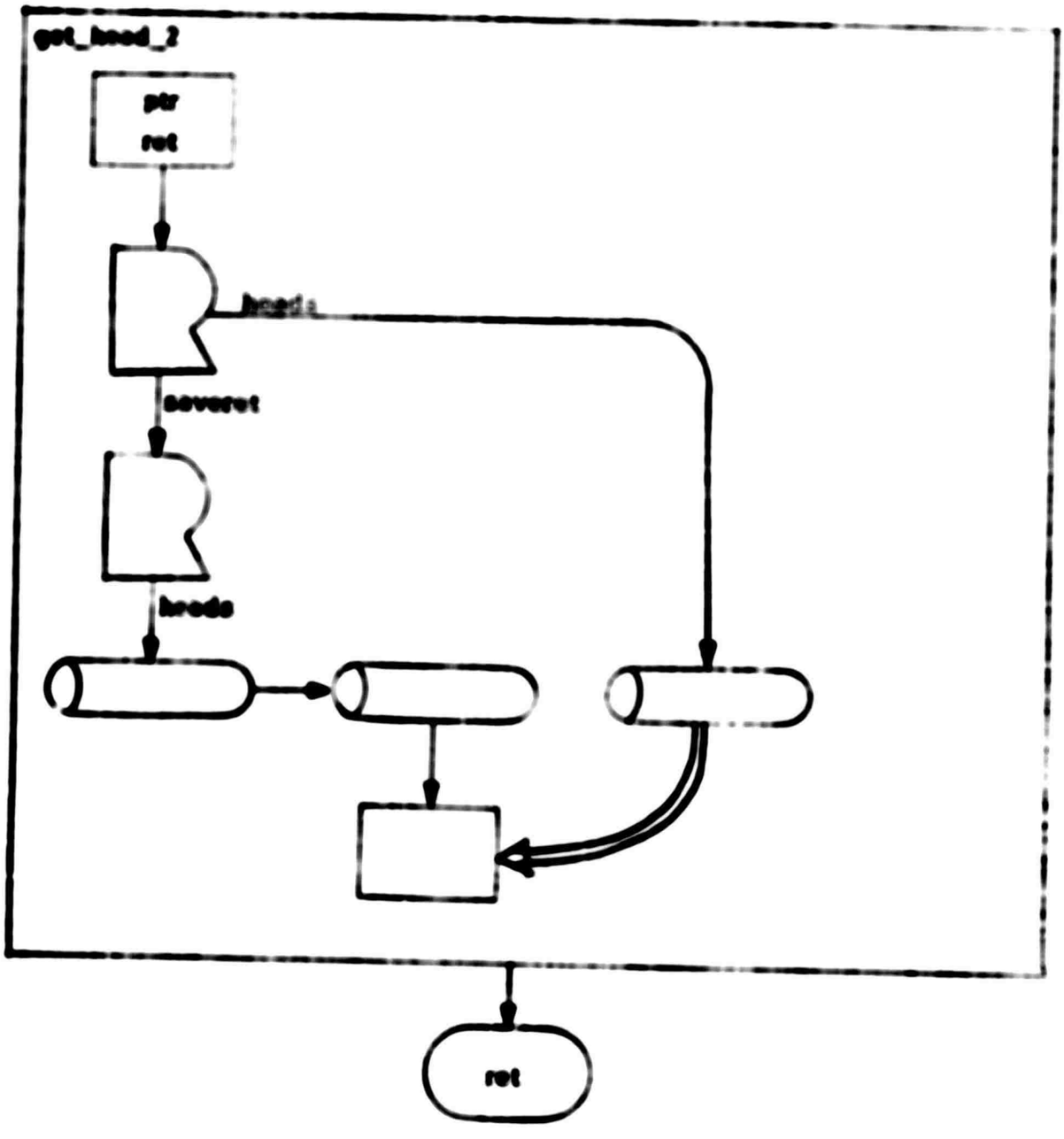

get_new_node
ptr
free
type
locate
charconn
end
head_l
ret

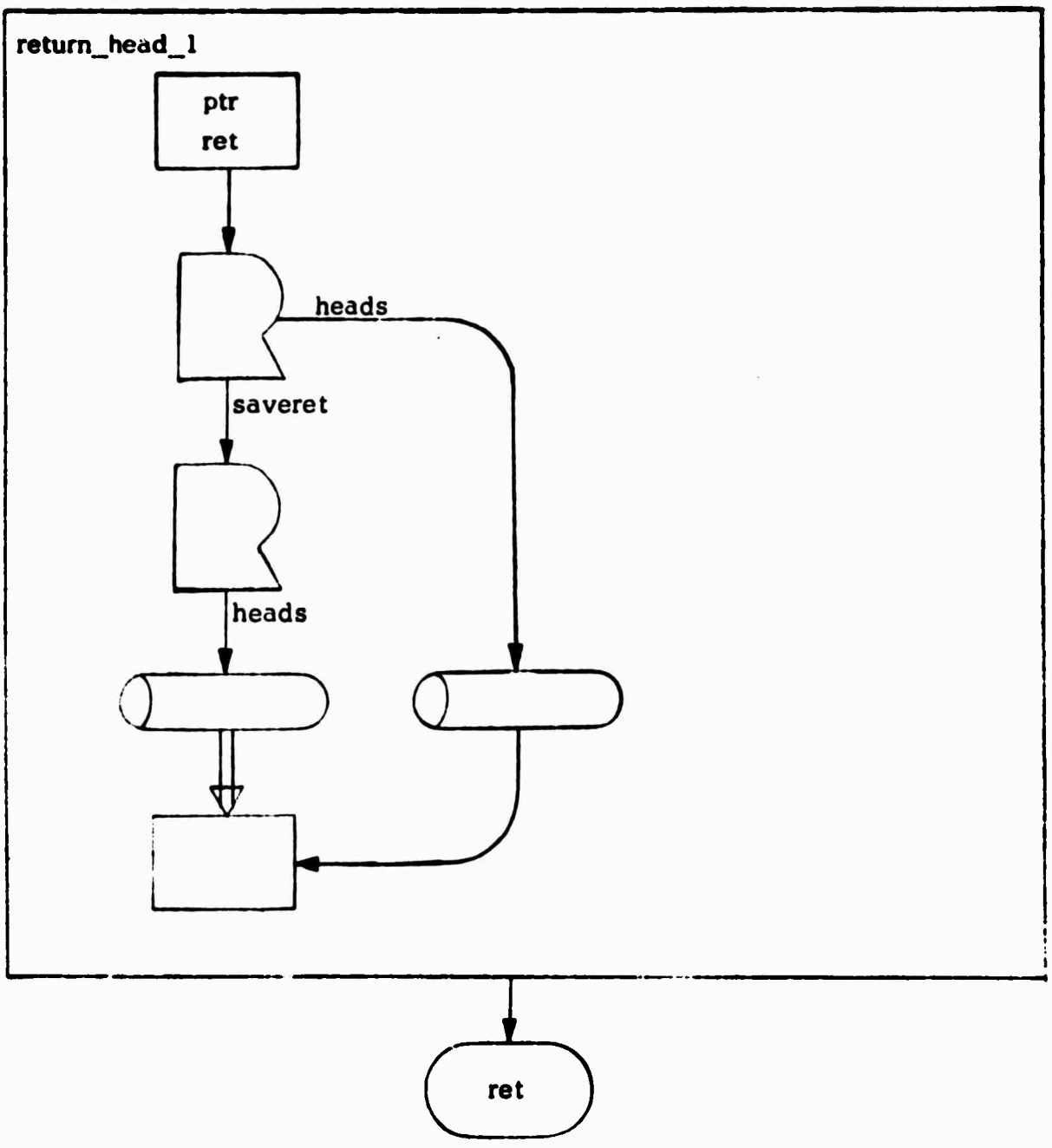

Procedure
Call_1
Send_1
port
port
port
Ret

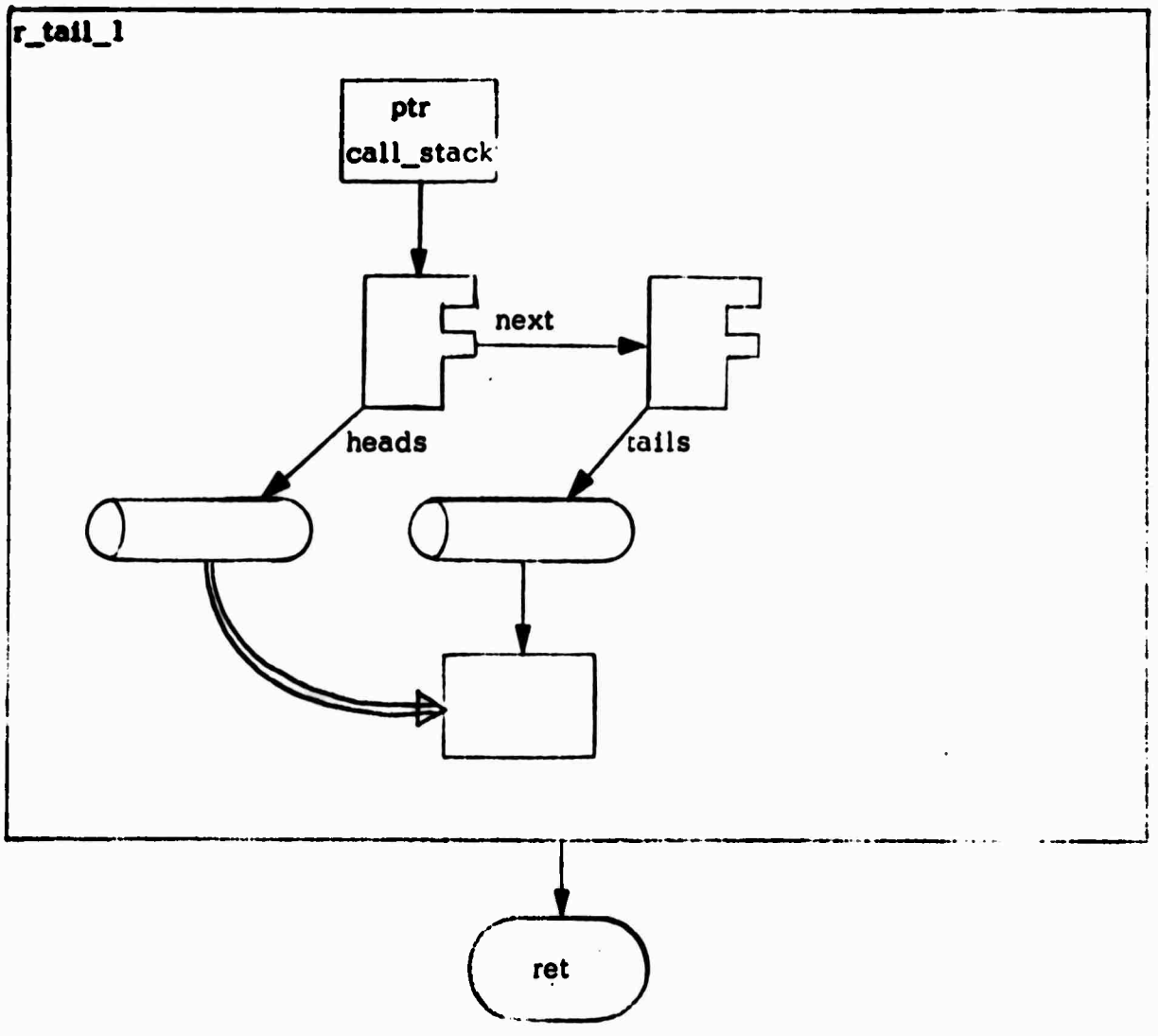

tail_1
ptr
ret
heads
several
tails
ret

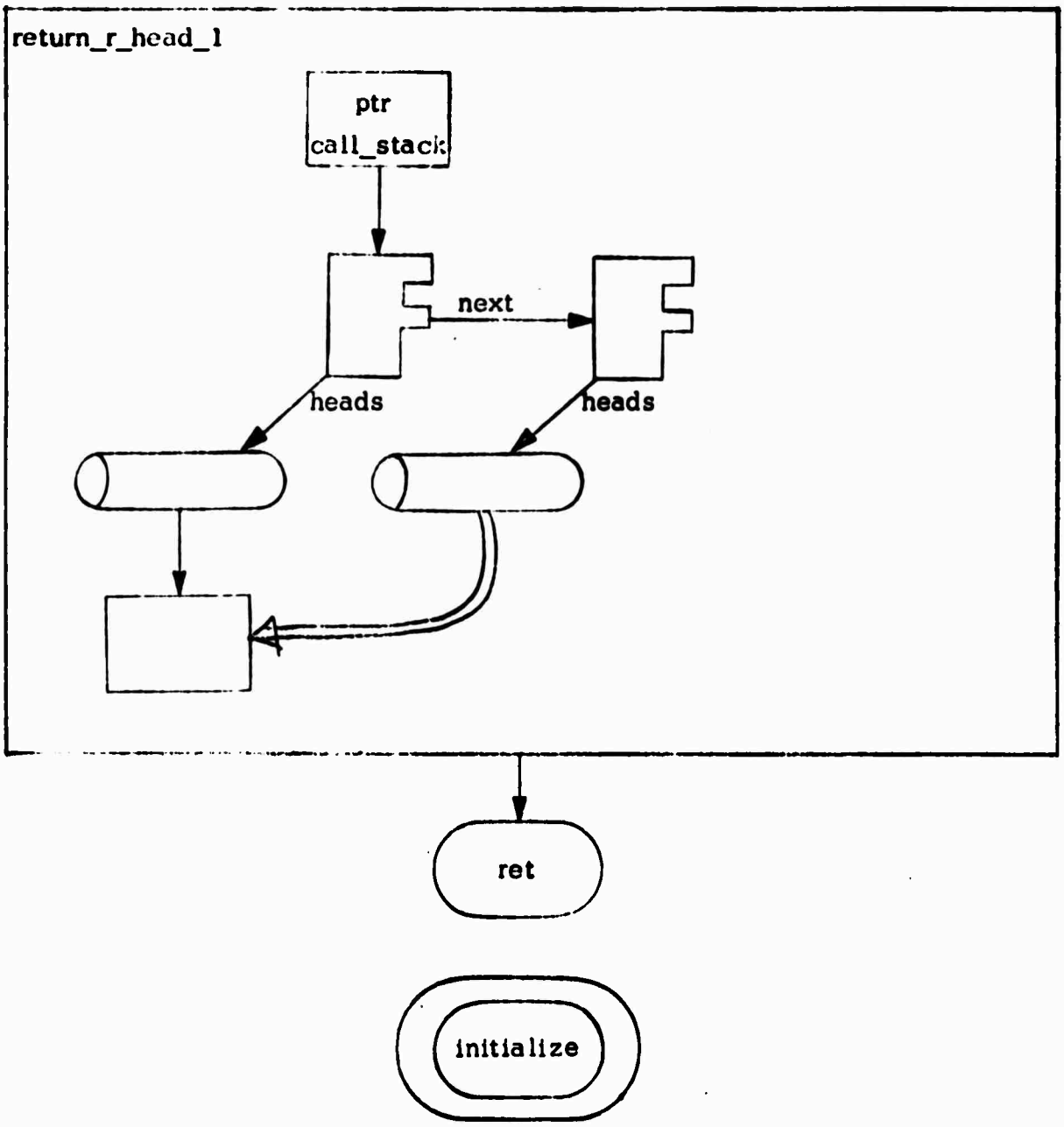

test_2
ptr
ret
savevret
ret


get_head_1
ptr
ret
T
heads
savecrt
heads
ret


get_hnod_2
ptr
ret
hnod_s
savesret
hnode
ret


return_head_1
ptr
ret
heads
saveret
heads
ret


r_tail_1
ptr
call_stack
next
heads
tails
ret


return_r_head_1
ptr
call_stack
next
heads
heads
ret
initialize